

```
printf("second child,  
%d\n", getpid());
```

**Broadview**  
www.broadview.com.cn

· 轻松学开发 ·

图解版

```
[root@yew ~]  
ftpboot># service xinetd restart  
nt gpio__open (struct inode *inode,  
struct file *filp)[r  
ot@yew ftpboot># service xinetd restartnt gpio  
__open (struct ino  
e *inode,  
struct file *filp)  
[root@yew ftpboot># see xinetd restart  
int gpio__open  
(str
```

# 轻松学

# C++

邹国华 编著

## 图解学编程，C++竟然这么简单

### 本书特点

- 654幅教学插图，轻松学习技术
- 159个典型示例，熟练掌握应用
- 729分钟视频，体验全新方式
- 87个课后题目，全面测试能力

### 随书DVD

729分钟全程视频 · 本书源代码 · PowerPoint电子课件

 **电子工业出版社**  
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY  
http://www.phei.com.cn

· 轻松学开发 ·



# 轻松学

# C++

---

邹国华 编著

电子工业出版社

Publishing House of Electronics Industry

北京•BEIJING



## 内容简介

本书由浅入深,全面、系统地介绍了 C++ 编程语言。本书最大的特色就是提供了大量的插图,一改过去编程书籍枯燥乏味的文字讲解,利用各种说明插图和运行结果示意图,生动形象地再现了 C++ 语言各项内容,使读者能够轻松地掌握学习内容。另外,作者专门为每一章编写了一些习题,以便读者对该章的学习水平进行检测。本书还录制了大量的配套教学视频,这些视频和书中的实例源代码一起收录于本书的配书光盘中。

本书共分 4 篇。第 1 篇是 C++ 概述篇,主要介绍 C++ 历史、开发环境搭建、编写第一个 C++ 程序等知识;第 2 篇是 C++ 基础语法篇,主要介绍数据的表示、语句、控制结构、数组、函数、指针、引用、复合数据类型等内容;第 3 篇是 C++ 面向对象篇,主要介绍类、对象、继承、派生、多态、运算符重载、类模板等内容;第 4 篇是 C++ 应用技术篇,详细介绍了输入/输出流、预处理、宏、标准模板库、程序调试、异常处理、文件等技术。

本书涉及面广,从基本操作到高级技术和核心原理,再到项目开发,几乎涉及 C++ 编程的所有重要知识。本书适合所有想全面学习 C++ 的人员阅读,也适合各种使用 C++ 开发的工程技术人员使用。对于经常使用 C++ 做开发的人员,更是一本不可多得的案头必备参考书。

未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有,侵权必究。

### 图书在版编目(CIP)数据

轻松学 C++ / 邹国华编著. — 北京: 电子工业出版社, 2013.5

(轻松学开发)

ISBN 978-7-121-19809-0

I. ①轻… II. ①邹… III. ①C 语言—程序设计 IV. ①TP312

中国版本图书馆 CIP 数据核字(2013)第 048269 号

策划编辑: 胡辛征

责任编辑: 高洪霞

特约编辑: 赵树刚

文字编辑: 马洪涛

印 刷:

装 订:

出版发行: 电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本: 860×1092 1/16 印张: 26.25 字数: 672 千字

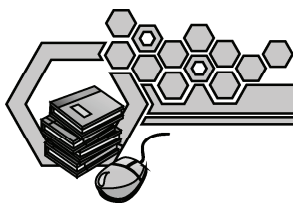
印 次: 2013 年 5 月第 1 次印刷

印 数: 4000 册 定价: 55.00 元(含 DVD 光盘 1 张)

凡所购买电子工业出版社图书有缺损问题, 请向购买书店调换。若书店售缺, 请与本社发行部联系, 联系及邮购电话: (010) 88254888。

质量投诉请发邮件至 [zltts@phei.com.cn](mailto:zltts@phei.com.cn), 盗版侵权举报请发邮件至 [dbqq@phei.com.cn](mailto:dbqq@phei.com.cn)。

服务热线: (010) 88258888。



从 20 世纪 80 年代 C++ 语言诞生以来,已经有二十多年了。在这期间, C++ 语言以简洁、高效的特点,广泛应用于不同的开发环境中。由于 C++ 的高运行效率和高开发效率,在不同的行业领域中都有卓越的表现。这些领域包括硬件、系统级的开发,也包括游戏、浏览器等普通软件。在当前网络应用爆发式发展中, C++ 语言作为核心开发语言之一将继续闪烁着璀璨的光芒。

笔者结合自己多年的 C++ 开发经验和心得体会,花费了一年多的时间编写本书。希望各位读者能在本书的引领下跨入 C++ 编程世界的大门,并成为一名开发高手。本书最大的特色就是结合大量的说明插图,全面、形象、系统、深入地介绍了 C++ 语言,并以大量实例贯穿于全书的讲解之中。学习完本书后,读者可以具备灵活应用 C++ 语言的能力。

## 本书特色

### 1. 大量教学插图,读书学习不再枯燥乏味

本书最大的特点就是通篇采用图片讲解,将传统的文字讲解转换为各种形式的图形图表,最大限度地提升读者的阅读兴趣,让读者在潜移默化中掌握 C++ 语言的开发精髓。

### 2. 配有大量多媒体语音教学视频,体验全新教学课堂

作者专门录制了大量的配套多媒体语音教学视频,以便让读者更加轻松、直观地学习本书内容,提高学习效率。这些视频与本书源代码一起收录于配书光盘中。

### 3. 讲解由浅入深,循序渐进,适合各个层次的读者阅读

本书从 C++ 语言的基础开始讲解,逐步深入到 C++ 语言的高级开发技术及应用,内容梯度从易到难,讲解由浅入深,循序渐进,适合各个层次的读者阅读。

### 4. 贯穿大量的开发实例和技巧,迅速提升开发水平

本书在讲解知识点时贯穿了大量短小精悍的典型实例,并给出了大量的开发技巧,以便让读者更好地理解各种概念和开发技术,体验实际编程,迅速提高开发水平。

## 本书内容及体系结构

### 第 1 篇 C++ 概述篇 (第 1 章)

本篇主要包括: C++ 发展历程、开发环境搭建、第一个 C++ 程序、程序结构等。通过本篇的学习,读者可以掌握 C++ 语言的特点、开发环境的配置和应用过程。

## 第2篇 C++基础语法篇（第2~8章）

本篇主要包括：数据类型、变量、常量、语句、控制结构、数组、函数、指针、引用、复合数据类型等内容。通过本篇的学习，读者可以基本掌握 C++面向过程编程的语法知识。

## 第3篇 C++面向对象篇（第9~13章）

本篇主要包括：类、对象、继承、派生、多态、运算符重载、类模板等技术。通过本篇的学习，读者可以掌握 C++面向对象开发的相关知识。

## 第4篇 C++应用技术（第14~18章）

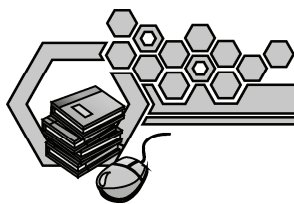
本篇主要包括：输入/输出流、预处理、宏、标准模板库 STL、程序调试、异常处理、文件等技术。通过本篇的学习，读者可以掌握 C++项目开发中的各项技术。

## 本书读者对象

- ☐ 从未接触过 C++的初学者。
- ☐ 了解一些 C++知识，希望进一步学习的自学者。
- ☐ 想学习一门技术，以方便找工作的求职者。
- ☐ C++开发爱好者。
- ☐ 大中专院校的学生和相关授课教师。
- ☐ 社会培训班学员。
- ☐ C++专业开发人员。
- ☐ 需要一本案头必备手册的程序员。

编著者

2013年1月



# 目 录

## 第 1 篇 C++概述篇

第 1 章 C++基础 .....	2
1.1 C++概述 .....	2
1.1.1 C++语言发展历程 .....	2
1.1.2 C++语言特点 .....	2
1.1.3 C++程序开发基本过程 .....	3
1.2 C++开发环境的搭建 .....	4
1.2.1 Visual C++ 6.0 安装 .....	4
1.2.2 Visual C++ 6.0 开发环境简介 .....	6
1.3 第一个 C++程序 .....	6
1.3.1 创建源程序 .....	6
1.3.2 编译链接 .....	9
1.3.3 调试运行 .....	10
1.3.4 典型 C++程序的执行过程 .....	12
1.3.5 使用 C++解决问题的流程 .....	12
1.4 C++程序的结构 .....	14
1.4.1 注释 .....	14
1.4.2 编译预处理和新旧标准 .....	14
1.4.3 程序主体 .....	15
1.5 小结 .....	15
1.6 习题 .....	15

## 第 2 篇 C++基础语法篇

第 2 章 数据的表示 .....	18
2.1 数 .....	18
2.1.1 二进制 .....	18
2.1.2 八进制 .....	19
2.1.3 十六进制 .....	20
2.2 数据的描述 .....	21
2.2.1 C++的构词方式——标识符 .....	21

2.2.2	特殊的标识符——关键字	22
2.2.3	整数类型	23
2.2.4	浮点类型	24
2.2.5	字符类型	25
2.2.6	布尔类型	27
2.3	变量	27
2.3.1	变量的声明和定义	28
2.3.2	不变的变量——常量	28
2.4	小结	29
2.5	习题	29
<b>第 3 章</b>	<b>程序的基本单位——语句</b>	<b>33</b>
3.1	语句的构成	33
3.1.1	表达式语句	33
3.1.2	输入/输出语句	33
3.2	运算符	35
3.2.1	运算符概述	35
3.2.2	运算符的分类	35
3.3	赋值运算符	36
3.3.1	赋值运算符——“=”	36
3.3.2	数据类型转换——隐式转换	37
3.3.3	显式转换	37
3.4	其他常用运算符	39
3.4.1	算术运算符	39
3.4.2	自增自减运算符	40
3.4.3	位运算符	41
3.4.4	复合赋值运算符	43
3.4.5	逗号运算符	43
3.4.6	sizeof 运算符	44
3.4.7	逻辑运算符	45
3.4.8	运算符的优先级和结合性	45
3.5	语句块	48
3.5.1	语句块的构成	48
3.5.2	作用域——变量的作用范围	48
3.6	小结	49
3.7	习题	49
<b>第 4 章</b>	<b>程序控制结构</b>	<b>56</b>
4.1	语句块的执行方式——顺序结构	56
4.2	条件的表达	56
4.2.1	单一条件的表达——关系运算符	56

4.2.2	多条件的表达——逻辑运算符 .....	57
4.3	选择结构 .....	58
4.3.1	最简单的选择——条件运算符 .....	58
4.3.2	单分支条件语句——if 语句 .....	59
4.3.3	双分支条件语句——if...else 语句 .....	61
4.3.4	多重 if...else 语句 .....	62
4.3.5	情况语句——switch 语句 .....	64
4.3.6	应用实例 .....	65
4.4	循环结构 .....	66
4.4.1	for 语句 .....	67
4.4.2	while 语句 .....	68
4.4.3	do...while 语句 .....	69
4.4.4	多重循环 .....	70
4.4.5	应用举例 .....	71
4.5	意外情况的表达——转向语句 .....	72
4.5.1	跳出语句——break 语句 .....	72
4.5.2	继续语句——continue 语句 .....	72
4.5.3	转向语句——goto .....	73
4.6	综合实例 .....	74
4.7	小结 .....	76
4.8	习题 .....	76
<b>第 5 章</b>	<b>数组 .....</b>	<b>81</b>
5.1	数组概述 .....	81
5.2	数组的来源 .....	82
5.3	一维数组 .....	83
5.3.1	一维数组的声明和定义 .....	83
5.3.2	一维数组的引用 .....	84
5.4	二维数组 .....	86
5.4.1	二维数组的声明和定义 .....	86
5.4.2	二维数组的初始化 .....	86
5.4.3	二维数组的引用 .....	88
5.4.4	多维数组在内存中如何排列元素 .....	89
5.5	字符串 .....	90
5.5.1	字符数组 .....	90
5.5.2	字符串的存储形式 .....	91
5.5.3	字符数组与字符串的区别 .....	91
5.5.4	字符串处理函数 .....	92
5.6	综合实例——杨辉三角 .....	94
5.7	小结 .....	97

5.8 习题	97
<b>第6章 函数</b>	<b>106</b>
6.1 函数概述	106
6.1.1 函数的基本概念	106
6.1.2 函数的分类	107
6.2 函数的组成	107
6.2.1 函数头	108
6.2.2 函数体	108
6.3 函数的声明和定义	110
6.3.1 函数原型——函数的声明	110
6.3.2 函数实现——函数的定义	111
6.4 函数参数传递	112
6.4.1 函数的形参和实参	112
6.4.2 值传递	113
6.5 函数的调用	115
6.5.1 函数的调用过程	115
6.5.2 无参函数的调用	115
6.5.3 带参函数的调用	116
6.5.4 默认形参值的调用	117
6.5.5 嵌套调用	120
6.5.6 数组作为函数参数	121
6.6 递归函数	123
6.6.1 直接递归	124
6.6.2 间接递归	125
6.7 main()函数	127
6.7.1 不带参数的 main()函数	127
6.7.2 带参数的 main()函数	127
6.8 函数的综合应用	128
6.9 小结	128
6.10 习题	128
<b>第7章 指针与引用</b>	<b>136</b>
7.1 指针概述	136
7.1.1 指针的基本概念	136
7.1.2 定义指针变量	137
7.1.3 初始化指针	138
7.2 指针的访问	139
7.2.1 指针的值	140
7.2.2 访问指针数据	140
7.2.3 小结指针 p	141

7.3	指针的算术运算.....	142
7.3.1	指针与整数的加减运算.....	142
7.3.2	指针加减 1 运算.....	143
7.3.3	指针的相减运算.....	144
7.4	特殊指针.....	144
7.4.1	数组指针.....	144
7.4.2	指向函数的指针——函数指针.....	145
7.4.3	指针数组.....	146
7.4.4	二级指针——指针的指针.....	147
7.4.5	多级指针——二级以上的指针.....	148
7.5	指针的应用.....	149
7.5.1	指向一维数组的指针.....	149
7.5.2	指向二维数组的指针.....	150
7.5.3	指向字符串的指针.....	152
7.5.4	指针作为函数参数.....	153
7.5.5	指针作为函数的返回值——指针函数.....	155
7.5.6	动态内存分配.....	156
7.6	引用.....	158
7.6.1	引用的应用.....	158
7.6.2	引用与指针.....	159
7.6.3	引用作为函数参数.....	160
7.7	小结.....	161
7.8	习题.....	162
<b>第 8 章</b>	<b>复合数据类型.....</b>	<b>169</b>
8.1	结构体.....	169
8.1.1	结构体概述.....	169
8.1.2	定义结构体类型.....	170
8.1.3	声明结构体变量.....	172
8.2	结构体的应用.....	173
8.2.1	初始化结构体变量.....	173
8.2.2	引用结构体变量成员.....	175
8.2.3	结构体指针.....	175
8.2.4	结构体数组.....	176
8.2.5	结构体和数组的比较.....	178
8.3	联合.....	178
8.3.1	定义联合类型.....	178
8.3.2	声明联合变量.....	179
8.3.3	引用联合类型成员.....	179
8.4	枚举.....	180



8.4.1	定义枚举类型 .....	180
8.4.2	声明枚举变量 .....	182
8.4.3	引用枚举变量成员 .....	182
8.5	用户自定义数据类型 .....	183
8.6	综合实例 .....	184
8.7	小结 .....	187
8.8	习题 .....	187

## 第 3 篇 C++面向对象篇

第 9 章	类和对象 .....	194
9.1	类和对象概述 .....	194
9.2	类和对象的基础语法 .....	195
9.2.1	类的声明 .....	195
9.2.2	实例化对象 .....	196
9.3	类的属性——数据成员 .....	196
9.3.1	类的属性的定义 .....	196
9.3.2	类的数据成员的特例——静态数据成员 .....	196
9.4	类的方法——成员函数 .....	197
9.4.1	类的方法的定义 .....	197
9.4.2	静态成员函数 .....	198
9.4.3	成员函数的类别（const 的另一种用法） .....	198
9.5	特殊的成员函数——构造函数和析构函数 .....	198
9.5.1	构造函数的概念 .....	198
9.5.2	构造函数的声明和定义 .....	200
9.5.3	构造函数的调用 .....	201
9.5.4	不带参数的构造函数 .....	202
9.5.5	带有默认参数的构造函数 .....	202
9.5.6	构造函数的重载 .....	203
9.5.7	特殊的构造函数——复制构造函数 .....	205
9.5.8	析构函数 .....	207
9.5.9	类和函数的联系 .....	208
9.5.10	this 指针 .....	209
9.6	小结 .....	209
9.7	习题 .....	209
第 10 章	继承与派生 .....	217
10.1	继承与派生的基础语法 .....	217
10.1.1	继承与派生概述 .....	217
10.1.2	声明派生类 .....	218

10.2	成员的访问 .....	219
10.2.1	类的成员的访问说明符 .....	219
10.2.2	类的成员的访问权限 .....	219
10.3	继承的访问控制 .....	219
10.3.1	私有继承 .....	220
10.3.2	公有继承 .....	221
10.3.3	保护继承 .....	223
10.3.4	特殊方法的继承——派生类的构造函数和析构函数 .....	225
10.4	多重继承 .....	227
10.4.1	声明多重继承 .....	227
10.4.2	二义性问题 .....	227
10.4.3	多重继承的构造函数和析构函数 .....	231
10.5	虚基类 .....	233
10.5.1	声明虚基类 .....	233
10.5.2	虚基类的构造函数和初始化 .....	234
10.6	友元 .....	235
10.6.1	友元的引入 .....	235
10.6.2	友元函数 .....	235
10.6.3	友元成员 .....	237
10.6.4	友元类 .....	239
10.7	综合实例 .....	240
10.8	小结 .....	244
10.9	习题 .....	244
<b>第 11 章</b>	<b>多态 .....</b>	<b>251</b>
11.1	多态概述 .....	251
11.1.1	什么是多态 .....	251
11.1.2	多态的引入 .....	252
11.1.3	联编 .....	253
11.2	函数重载 .....	254
11.3	虚函数 .....	255
11.3.1	定义虚函数 .....	256
11.3.2	多级继承和虚函数 .....	258
11.4	纯虚函数与抽象类 .....	259
11.4.1	纯虚函数 .....	259
11.4.2	抽象类 .....	261
11.5	综合实例 .....	263
11.6	小结 .....	264
11.7	习题 .....	264

<b>第 12 章 运算符重载</b> .....	269
12.1 运算符重载概述.....	269
12.1.1 什么是运算符重载.....	269
12.1.2 运算符重载的特点.....	270
12.2 运算符重载形式.....	271
12.2.1 运算符重载为类的成员函数.....	272
12.2.2 运算符重载为类的友元函数.....	275
12.2.3 运算符成员函数与友元运算符函数的比较.....	278
12.3 特殊运算符重载.....	278
12.3.1 “++”和“--”重载.....	278
12.3.2 赋值运算符“=”重载.....	280
12.3.3 下标运算符“[]”重载.....	281
12.4 类类型转换.....	283
12.5 小结.....	284
12.6 习题.....	284
<b>第 13 章 类模板</b> .....	290
13.1 什么是类模板.....	290
13.2 定义类模板.....	292
13.2.1 语法.....	292
13.2.2 非类型参数.....	294
13.2.3 模板参数的默认实参.....	294
13.3 生成类模板的实例.....	295
13.3.1 类型参数的模板实例化.....	295
13.3.2 非类型参数的模板实例化.....	296
13.3.3 类模板示例.....	296
13.4 类模板的静态成员.....	298
13.5 类模板的友元.....	300
13.5.1 非模板的友元类和友元函数.....	301
13.5.2 与模板参数不绑定的友元类和友元函数模板.....	301
13.5.3 与模板参数绑定的友元类和友元函数模板.....	302
13.6 类模板的特化.....	303
13.6.1 类模板的全特化.....	303
13.6.2 类模板的偏特化.....	304
13.6.3 类模板的匹配规则.....	304
13.7 小结.....	305
13.8 习题.....	305

## 第 4 篇 C++应用技术篇

第 14 章 输入/输出流.....	308
14.1 输入/输出流的引入.....	308
14.1.1 C 语言中的输入/输出缺陷.....	308
14.1.2 输入/输出流简介.....	309
14.1.3 输入/输出流类层次.....	310
14.2 标准输入/输出流.....	311
14.2.1 标准输出流对象.....	312
14.2.2 标准输入流对象.....	312
14.3 输入/输出流成员函数.....	313
14.3.1 get()函数.....	313
14.3.2 getline()函数.....	314
14.3.3 put()函数.....	315
14.3.4 read()和 write()函数.....	315
14.3.5 其他成员函数.....	316
14.4 输入/输出格式控制.....	317
14.4.1 用 ios 类的成员函数进行格式控制.....	317
14.4.2 使用格式控制符进行格式控制.....	320
14.5 用户自定义数据类型的输入/输出.....	322
14.5.1 重载输出运算符“<<”.....	322
14.5.2 重载输入运算符“>>”.....	323
14.6 命名空间.....	325
14.6.1 命名空间概述.....	325
14.6.2 定义命名空间.....	325
14.6.3 使用命名空间.....	326
14.7 小结.....	327
14.8 习题.....	327
第 15 章 预处理和宏.....	331
15.1 预处理概述.....	331
15.2 宏.....	331
15.2.1 宏展开.....	332
15.2.2 替代常量.....	332
15.2.3 替代运算符.....	334
15.3 带参数的宏.....	334
15.3.1 定义带参数的宏.....	335
15.3.2 注意宏展开的结果.....	336
15.3.3 带参数的宏与函数的比较.....	338
15.4 条件编译.....	338

15.4.1	宏指令.....	338
15.4.2	使用条件编译.....	340
15.5	文件包含和头文件卫士.....	342
15.5.1	包含文件指令.....	342
15.5.2	搜索头文件.....	343
15.5.3	头文件卫士.....	344
15.6	预定义的宏.....	346
15.7	小结.....	347
15.8	习题.....	347
<b>第 16 章</b>	<b>标准模板库.....</b>	<b>349</b>
16.1	标准模板库概述.....	349
16.1.1	C++标准库.....	349
16.1.2	STL 的形成.....	350
16.1.3	STL 的组成.....	350
16.1.4	STL 的引入.....	351
16.2	算法.....	353
16.3	容器.....	354
16.3.1	容器概述.....	355
16.3.2	向量.....	355
16.3.3	列表.....	356
16.3.4	集合.....	358
16.3.5	双端队列.....	358
16.3.6	栈.....	360
16.3.7	映射和多重映射.....	361
16.4	迭代器.....	361
16.5	小结.....	363
16.6	习题.....	363
<b>第 17 章</b>	<b>程序调试与异常处理.....</b>	<b>365</b>
17.1	程序错误.....	365
17.1.1	编译错误.....	365
17.1.2	逻辑错误.....	366
17.1.3	运行错误.....	367
17.1.4	程序调试.....	368
17.2	异常处理.....	369
17.2.1	基本思想.....	369
17.2.2	抛出异常.....	370
17.2.3	捕获异常.....	371
17.2.4	自定义异常对象.....	373
17.3	异常处理实例.....	374

17.4 小结 .....	375
17.5 习题 .....	375
<b>第 18 章 文件 .....</b>	<b>378</b>
18.1 文件概述 .....	378
18.1.1 操作文件的过程 .....	379
18.1.2 处理文件流的类 .....	381
18.2 文件的打开与关闭 .....	384
18.2.1 打开文件 .....	384
18.2.2 关闭文件 .....	385
18.3 文件的顺序读/写 .....	386
18.3.1 读/写文本文件 .....	386
18.3.2 读/写二进制文件 .....	387
18.4 文件的随机读/写 .....	391
18.5 小结 .....	394
18.6 习题 .....	394

## 第 1 篇 C++概述篇

# 第 1 章 C++ 基础

C++是功能强大的开发语言，它是在 C 语言的基础上增加了面向对象程序设计的要素而发展起来的。本章将简要介绍 C++的发展历程及特点，着重介绍 C++的编译环境及使用该环境进行第一个 C++程序的设计。

## 1.1 C++概述

C++语言是一种应用较广的面向对象的程序设计语言。本节主要讲解 C++的发展、特点及其程序开发基本过程。

### 1.1.1 C++语言发展历程

C++语言起源于 C 语言。1980 年，美国 Bjarne Stroustrup 及其同事从 Simula67 中引入面向对象的特征，开发出一种程序设计语言。起名为“带类的 C”，至 1983 年改名为 C++，随后的发展如图 1-1 所示。

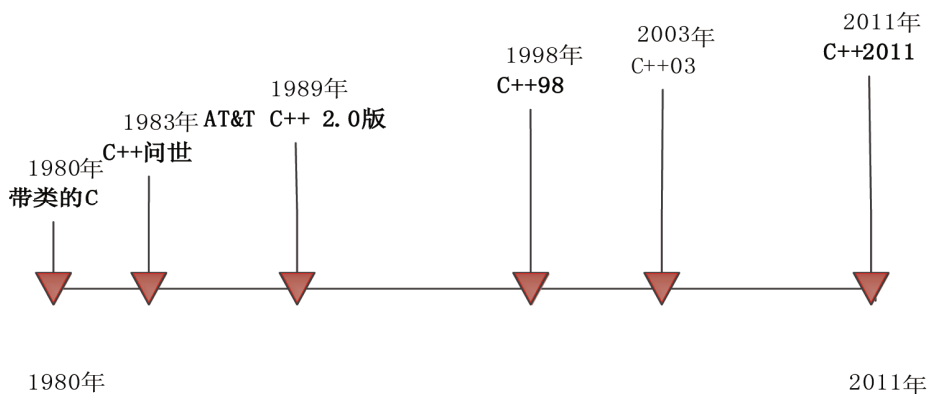


图 1-1 C++语言发展

1998 年，C++标准委员成立，随后发布了 C++ 98、C++ 03、C++ 2011 三个标准。

### 1.1.2 C++语言特点

C++具有两方面的特点：其一，C++是 C 语言的超集，因此能与 C 语言兼容；其二，C++支持面向对象的程序设计，具体有封装性、继承性、多态性三个特点，如图 1-2 所示。



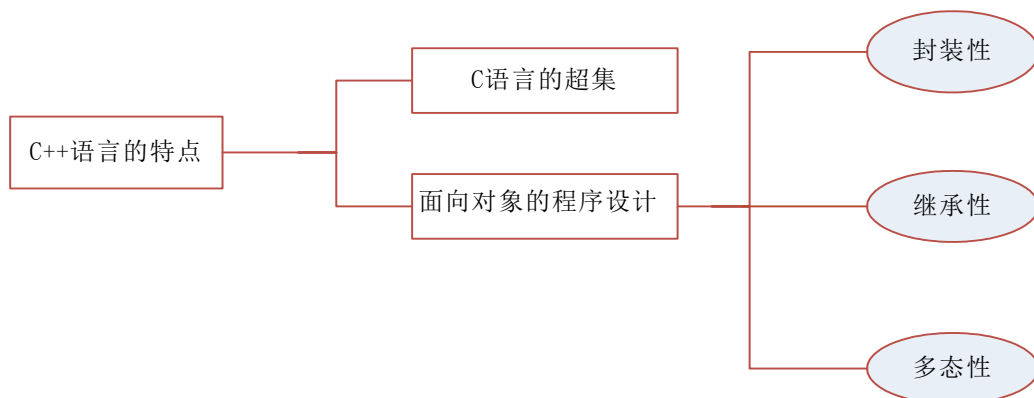


图 1-2 C++语言的特点

C++既可以支持面向过程的程序设计，也可进行面向对象的程序设计。下面来讲一下什么是面向过程和面向对象。

### 1. 面向过程的程序设计

在 20 世纪 60 年代计算机发展的初期，计算机和编程是少数聪明人的玩具。程序员可以根据自己的喜好随心所欲地进行程序设计。大多数程序代码组织混乱，可以说只有程序员本人可以看懂。随着计算机的发展和程序规模的扩大，凸显出了很多问题。这些问题使程序质量低下，进度缓慢，预算严重超支。为此，人们提出了结构化程序设计方法，探讨了面向过程的 3 种基本结构，即顺序、分支和循环。

面向过程的程序设计方法对于规模较小的软件是适用的，但当软件规模达到一定程度时，这种程序设计方法就显现出了稳定性低，以及可修改性和可重用性差的弊端。

### 2. 面向对象的程序设计

面向对象的程序设计与结构化的程序不同。由 C++编写的结构化的程序是由一个个函数组成，而由 C++编写的面向对象的程序是由一个个对象组成的，对象之间通过消息而相互作用。

在结构化的程序设计中，我们要解决某一个问题，就是要确定这个问题能够分解为哪些函数，数据能够分解为哪些基本类型。也就是说，思考方式是面向机器结构的，而不是面向问题结构的，需要在问题结构和机器结构之间建立联系。面向对象的程序设计方法的思考方式是面向问题结构的，它认为现实世界是由对象组成的。面向对象的程序设计方法解决某个问题，要确定这个问题是由哪些对象组成的，对象之间的相互关系是什么。

## 1.1.3 C++程序开发基本过程

了解 C++程序开发的基本过程，可以使读者更好地理解 C++程序的运行情况。C++程序开发基本过程如图 1-3 所示。

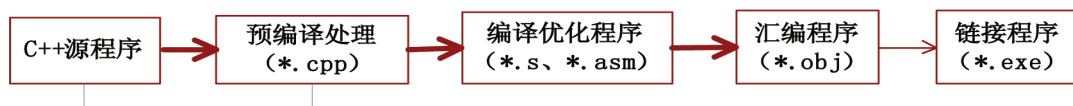


图 1-3 C++程序开发基本过程

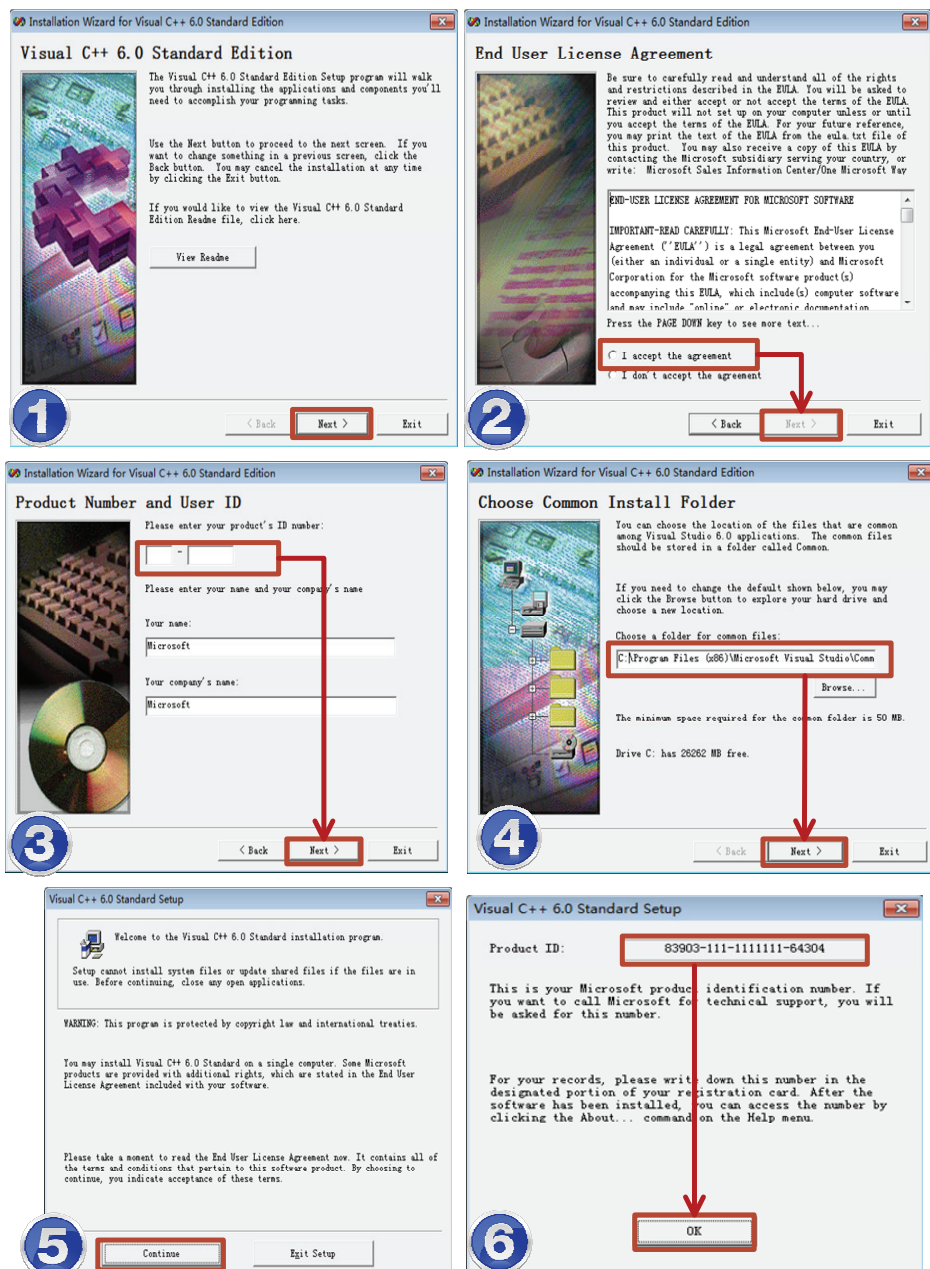


## 1.2 C++开发环境的搭建

Visual C++ 6.0 是运行 C++ 的集成环境，目前在国内使用比较广泛。本节主要讲解 Visual C++ 6.0 的环境搭建。

### 1.2.1 Visual C++ 6.0 安装

在 Windows 7 环境下，将 Visual C++ 6.0 的压缩包解压后，再运行其中的 Setup 应用程序文件即可开始安装。因安装过程比较复杂，读者需认真根据如图 1-4 所示的安装过程进行安装。



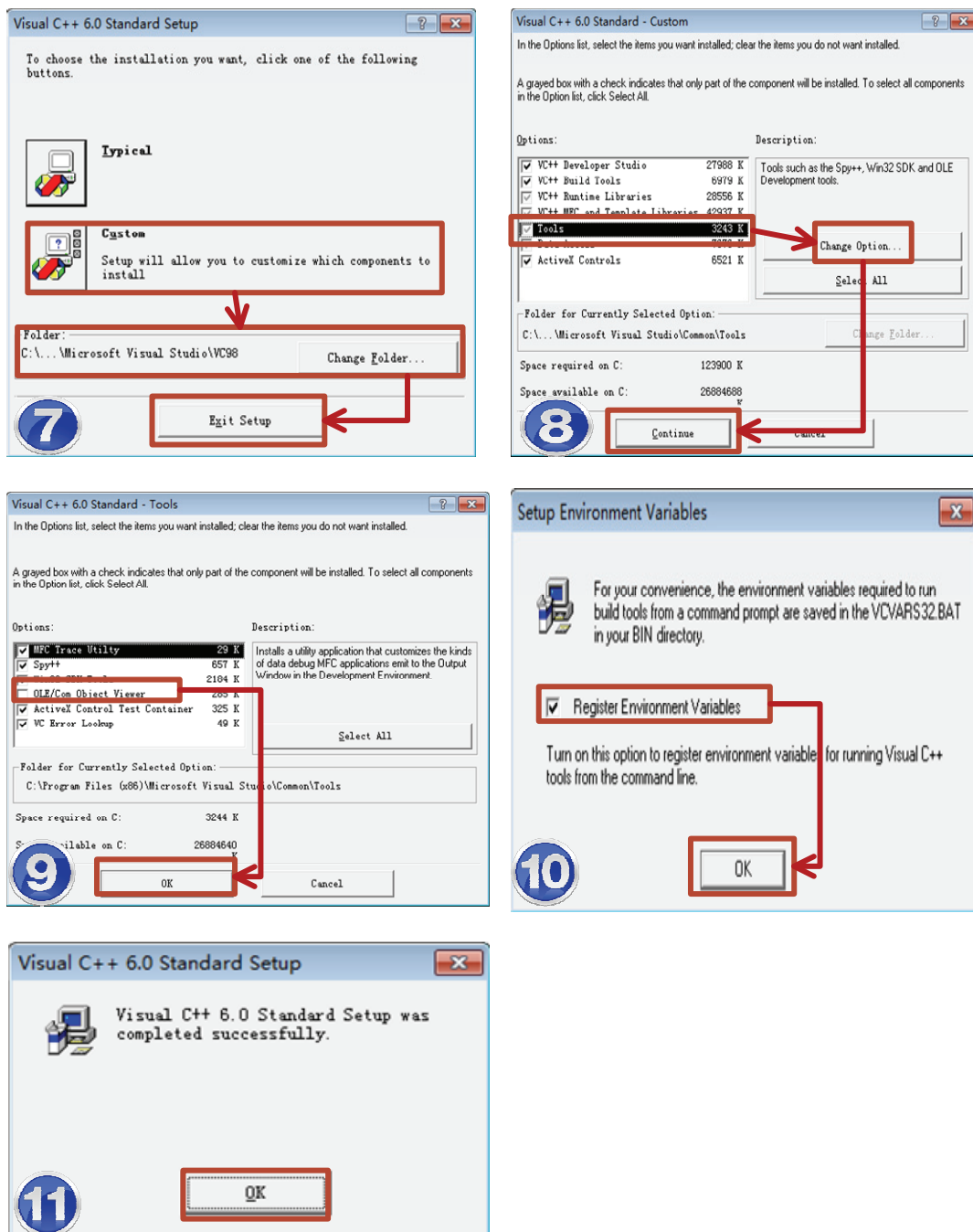


图 1-4 Visual C++ 6.0 的安装过程



**注意：**（1）对话框 3 中的序列号需要读者自己在微软购买。

（2）对话框 7 中提供了两种安装模式：Typical（典型）和 Custom（自定义）模式。在 Windows 7 下，必须使用 Custom 模式。然后在对话框 8 中勾选“Tools”复选框，并单击“Change Option”按钮，不安装 OLE 选项。



## 1.2.2 Visual C++ 6.0 开发环境简介

Visual C++ 6.0（简称 VC6.0）是一个功能强大的可视化集成开发工具。Visual C++一般可分为 3 个版本：学习版、专业版和企业版，不同的版本适合于不同类型的应用程序开发，本书用的是企业版。Visual C++ 6.0 的工作界面如图 1-5 所示。

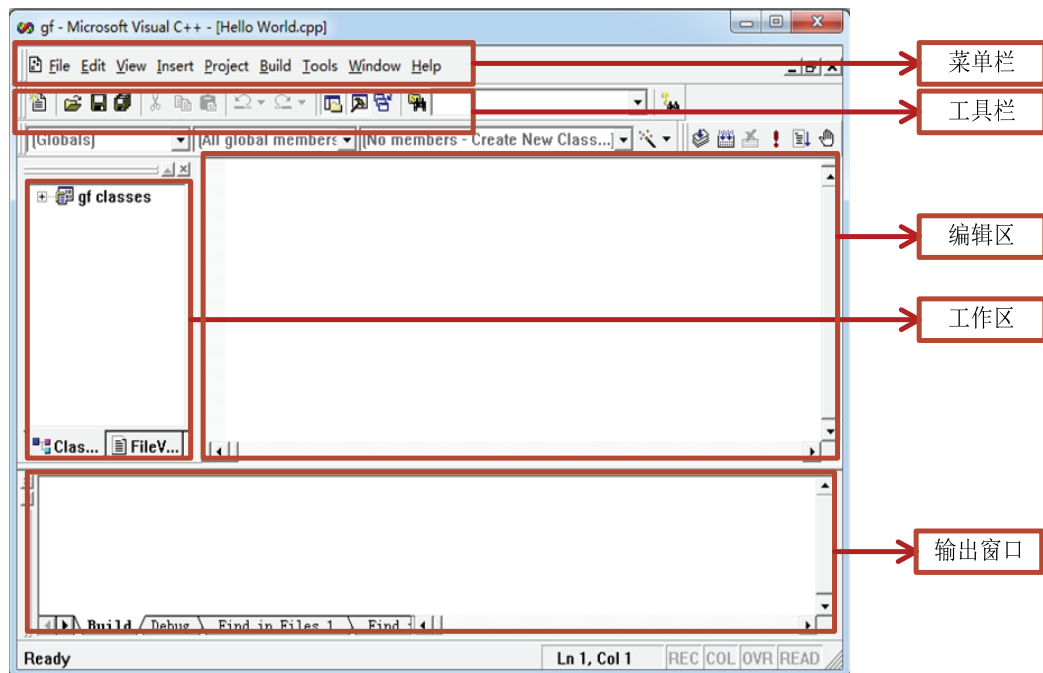


图 1-5 Visual C++ 6.0 工作界面

工作区用于显示开发项目中的各部分内容；编辑区用于对代码或资源进行操作；输出窗口会给出用户多种提示信息。



## 1.3 第一个 C++ 程序

为了让读者更好地理解，本节给出第一个 C++ 程序代码“Hello World”，以及其在 Visual C++ 6.0 中的编译、连接和运行步骤。

### 1.3.1 创建源程序

打开 Visual C++ 6.0 的集成开发环境。选择“File”|“New”命令，弹出“New”对话框，如图 1-6 所示。新建一个 Win32 Console Application 工程，工程名、存储位置读者可以自己选择。单击“OK”按钮后弹出如图 1-7 所示的对话框。

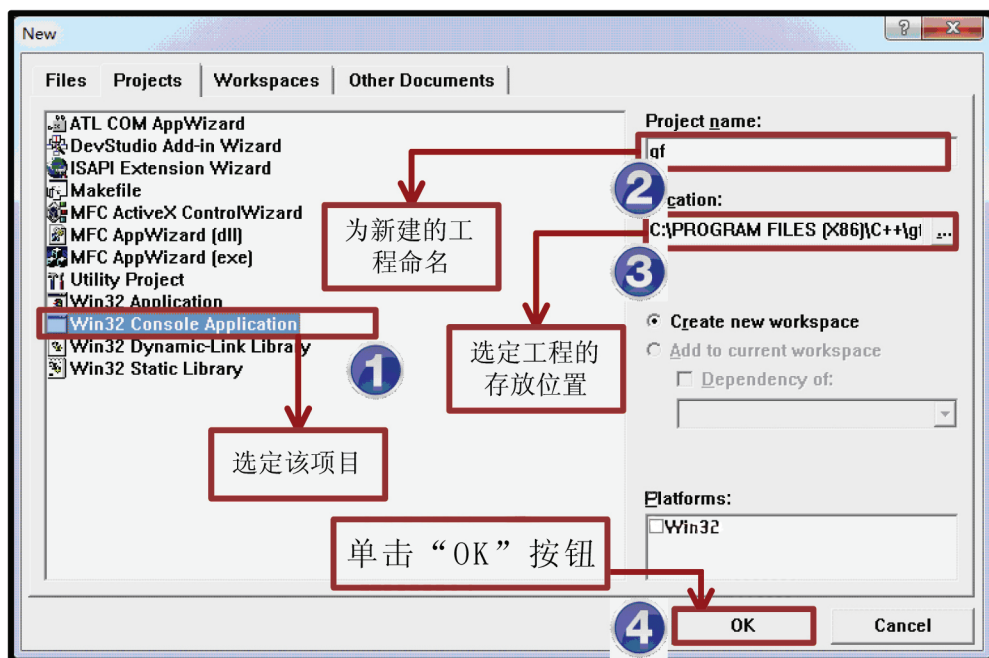


图 1-6 “new”对话框

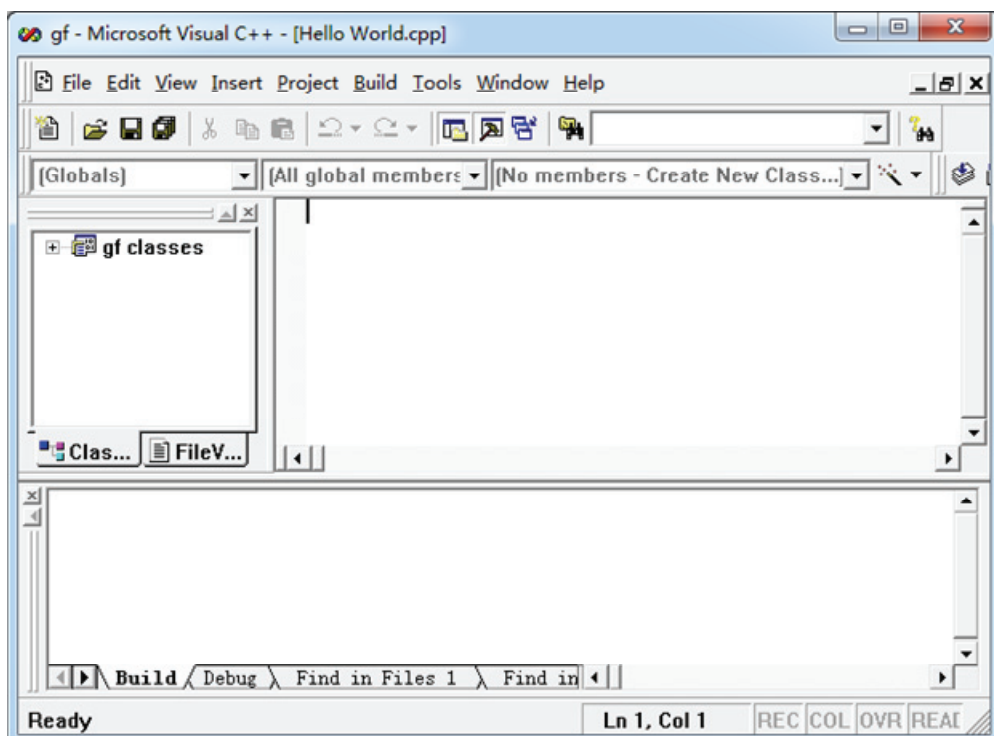


图 1-7 Visual C++ 6.0 集成开发环境

工程建立后即可创建源程序，具体步骤如下：

(1) 选择“File”|“New”命令，弹出“New”对话框，如图 1-8 所示。

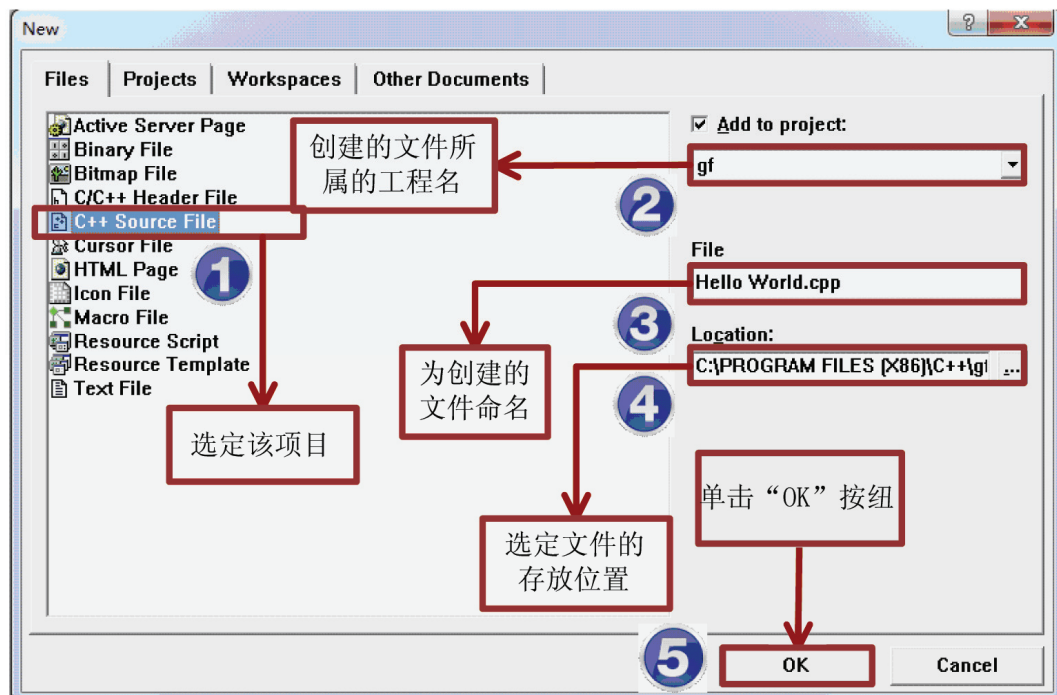


图 1-8 “New” 对话框

(2) 选择“File”选项卡，选择其中的“C++ Source File”选项，并在右侧输入文件名及路径。在该示例中，文件名为“Hello World”，选择路径后，单击“OK”按钮，结果如图 1-9 所示。

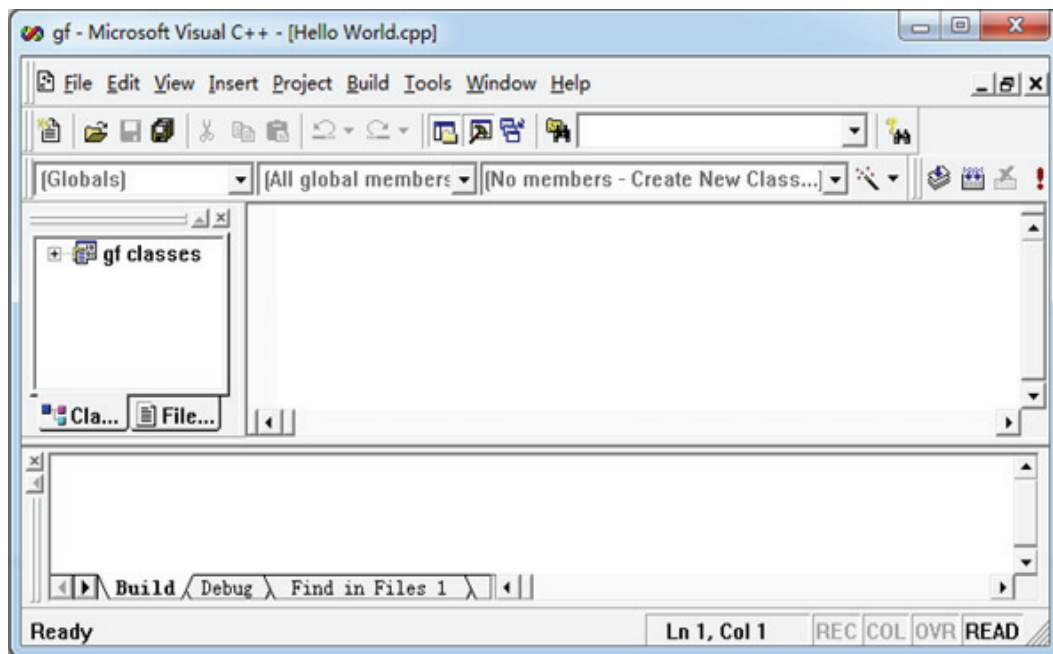


图 1-9 新建 C++源文件





(3) 在编辑区中输入如图 1-10 所示的程序代码。

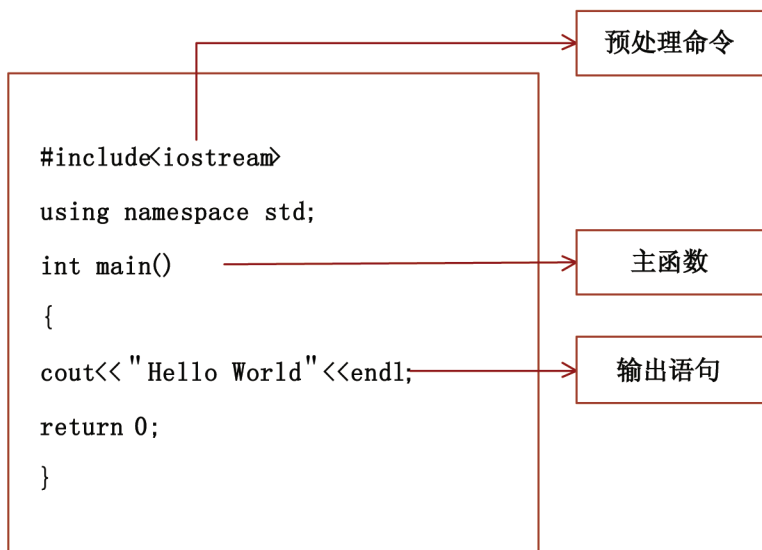


图 1-10 代码

输入完成后，效果如图 1-11 所示。

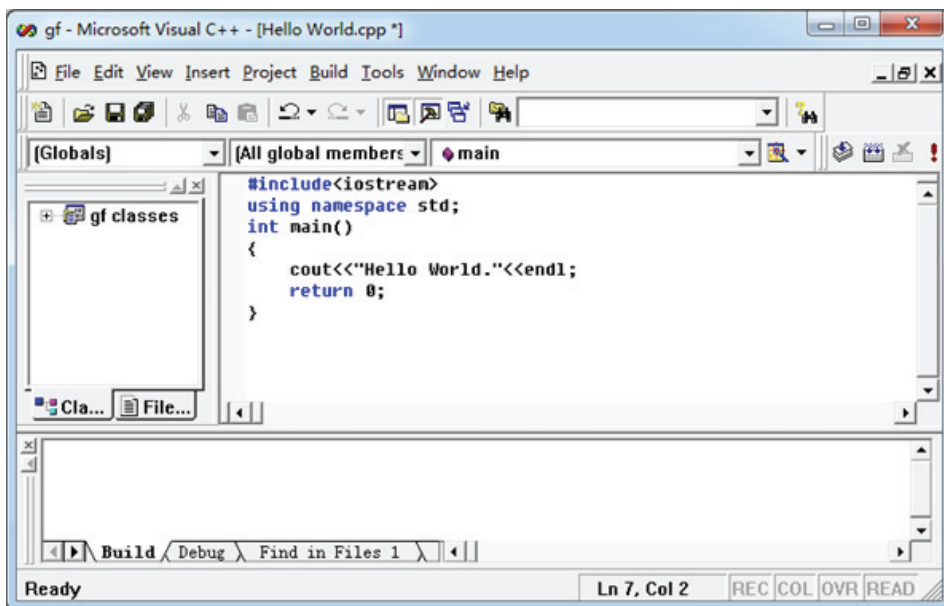


图 1-11 编辑源文件

### 1.3.2 编译链接

创建上述源程序后先保存，再对该源程序进行编译，使用“Build”|“Compile”命令或按快捷键 Ctrl+F7。编译结果如图 1-12 所示。编译没有错误后对其进行链接，使用“Build”|“Build”命令或按快捷键 F7，以建立可执行文件（.exe 文件），连接结果如图 1-13 所示。

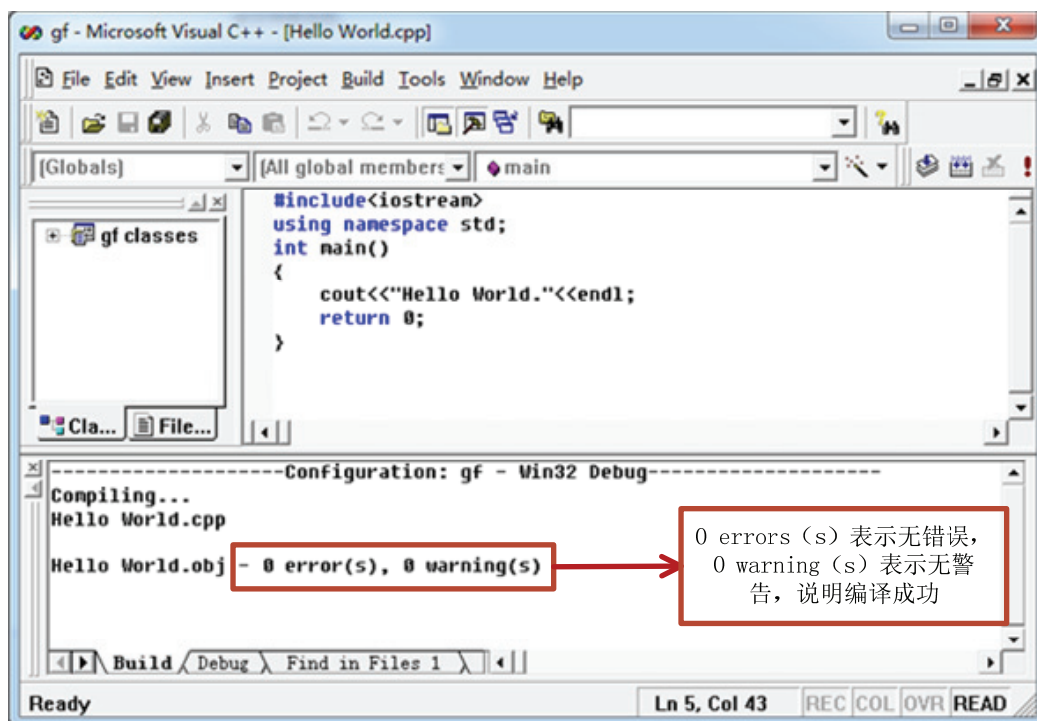


图 1-12 编译结果

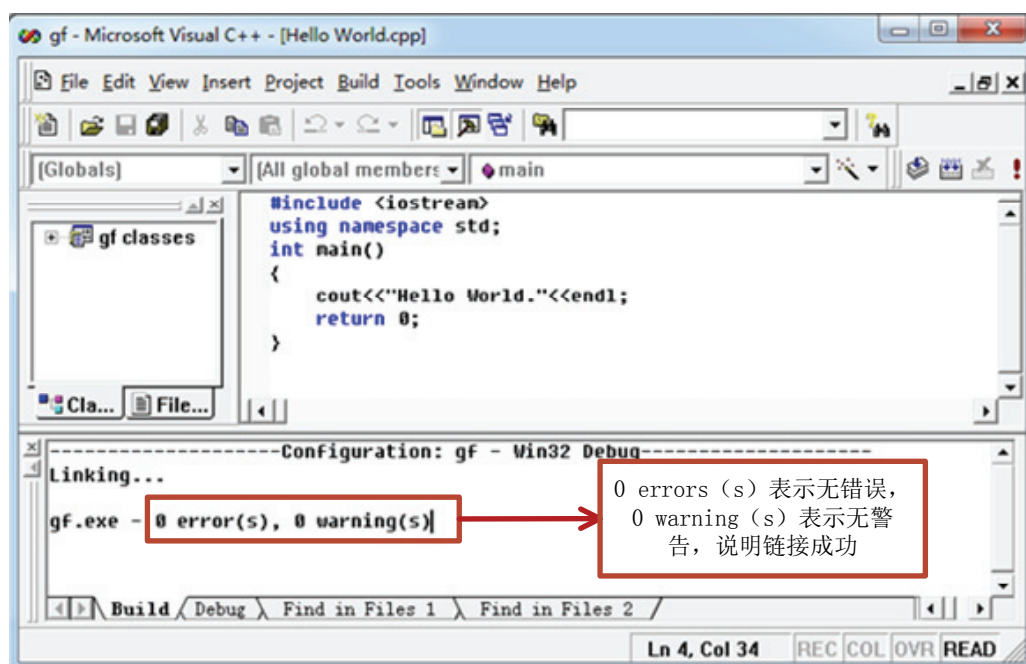


图 1-13 链接结果

### 1.3.3 调试运行

通过编译链接后, 选择“Build” | “Execute”命令或按快捷键 Ctrl+F5 运行源程序, 运行





结果如图 1-14 所示。

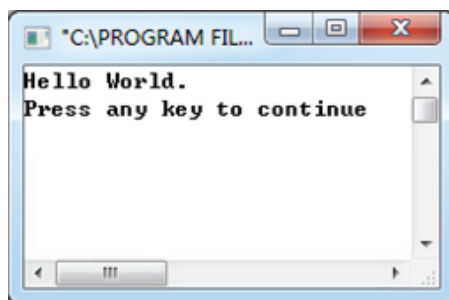


图 1-14 运行结果

程序运行完，关闭 VC 之后，若下一次打开 VC 要对 Hello.world 文件进行修改，则需要知道文件的存放位置，以及哪个文件才是我们要找的。下面就以本例进行说明。本例中创建的工程位置在“E:\C++\gf”，此时工程文件夹 gf 存放的文件类型及其说明如图 1-15 所示。

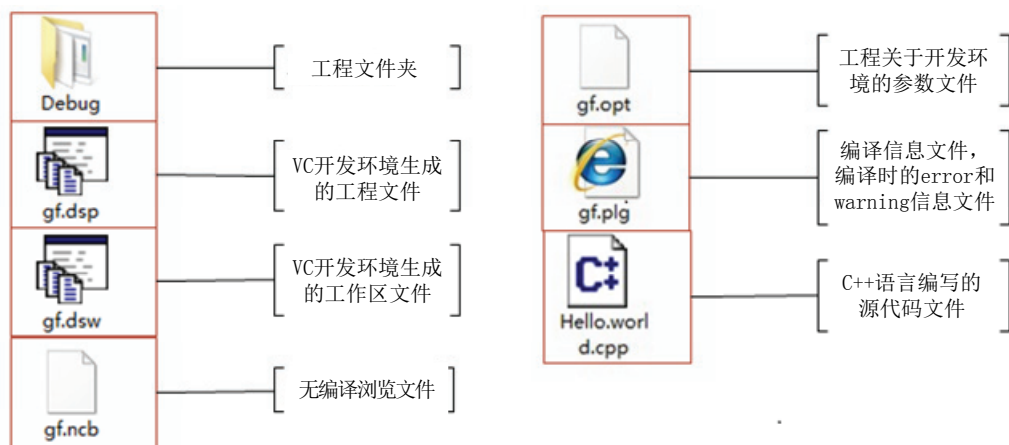


图 1-15 工程 gf 中的文件类型

将图 1-15 中的 Debug 文件夹打开后，其中的文件类型及其介绍如图 1-16 所示。

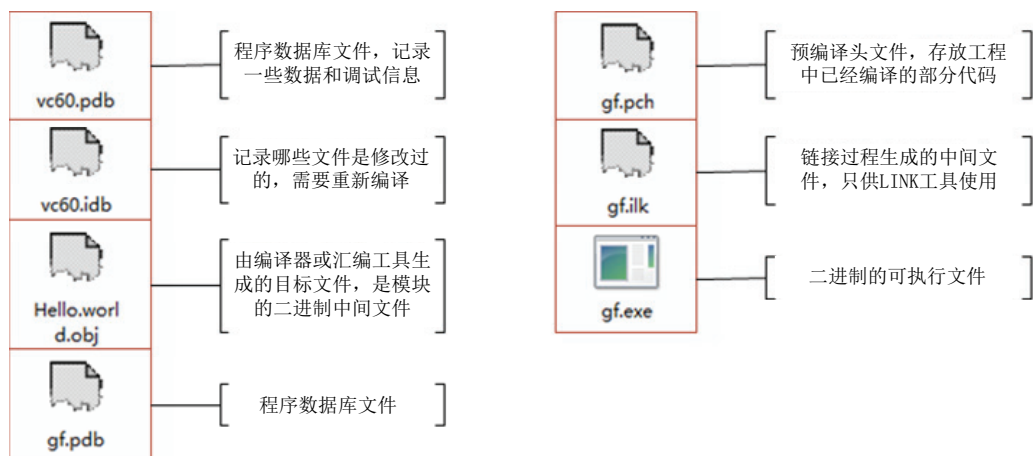


图 1-16 Debug 文件夹中的文件



在上述所有与 C++ 有关的文件中，.cpp 最为重要，程序的源代码就保存在此文件中，其他文件都是源文件在编译和链接时产生的相关临时文件。

### 1.3.4 典型 C++ 程序的执行过程

典型 C++ 程序的执行过程如图 1-17 所示。

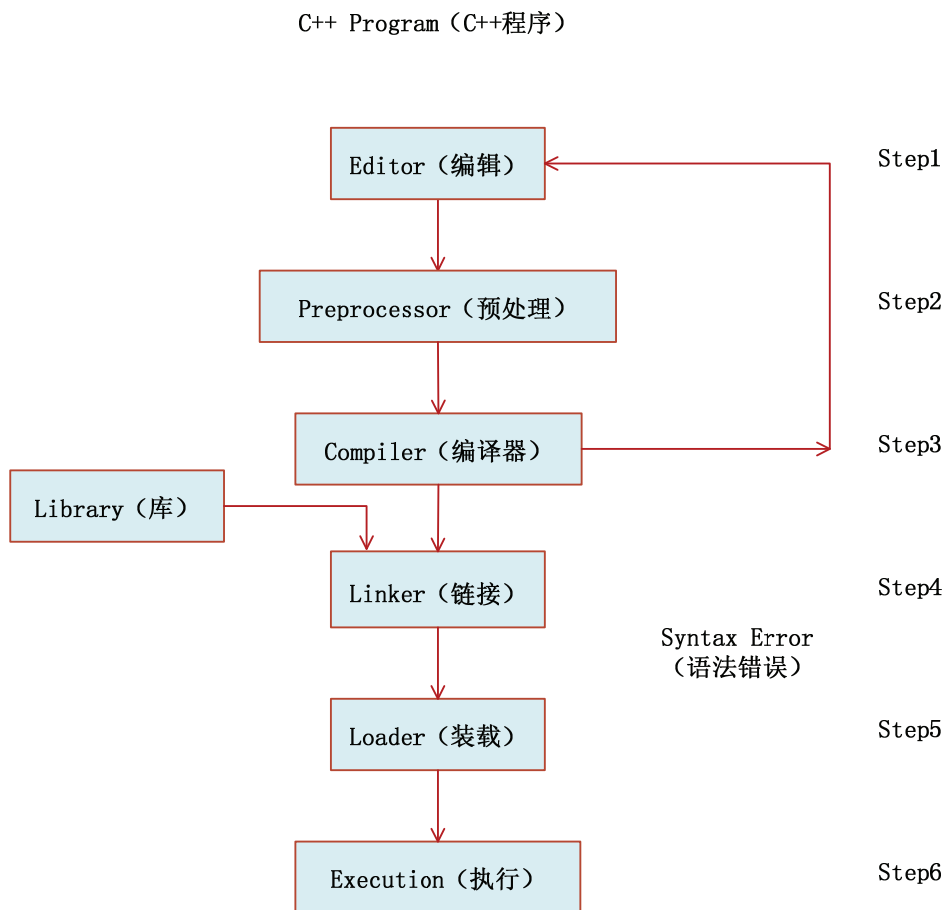


图 1-17 执行一个 C++ 程序

在 VC 中，读者只需要在代码编辑窗口中输入代码，其他诸如预处理、编译、链接、装载、执行等由 VC 来完成，用户无须关心。

### 1.3.5 使用 C++ 解决问题的流程

使用 C++ 解决问题的流程如下：要解决一个实际问题，首先程序员要对问题进行分析，确定算法（即解决问题的方法），然后再考虑算法在机器上的物理实现；接下来将整理好的代码输入到计算机中，输入完成后进行编译、链接、运行，最后得出结果，如图 1-18 所示。

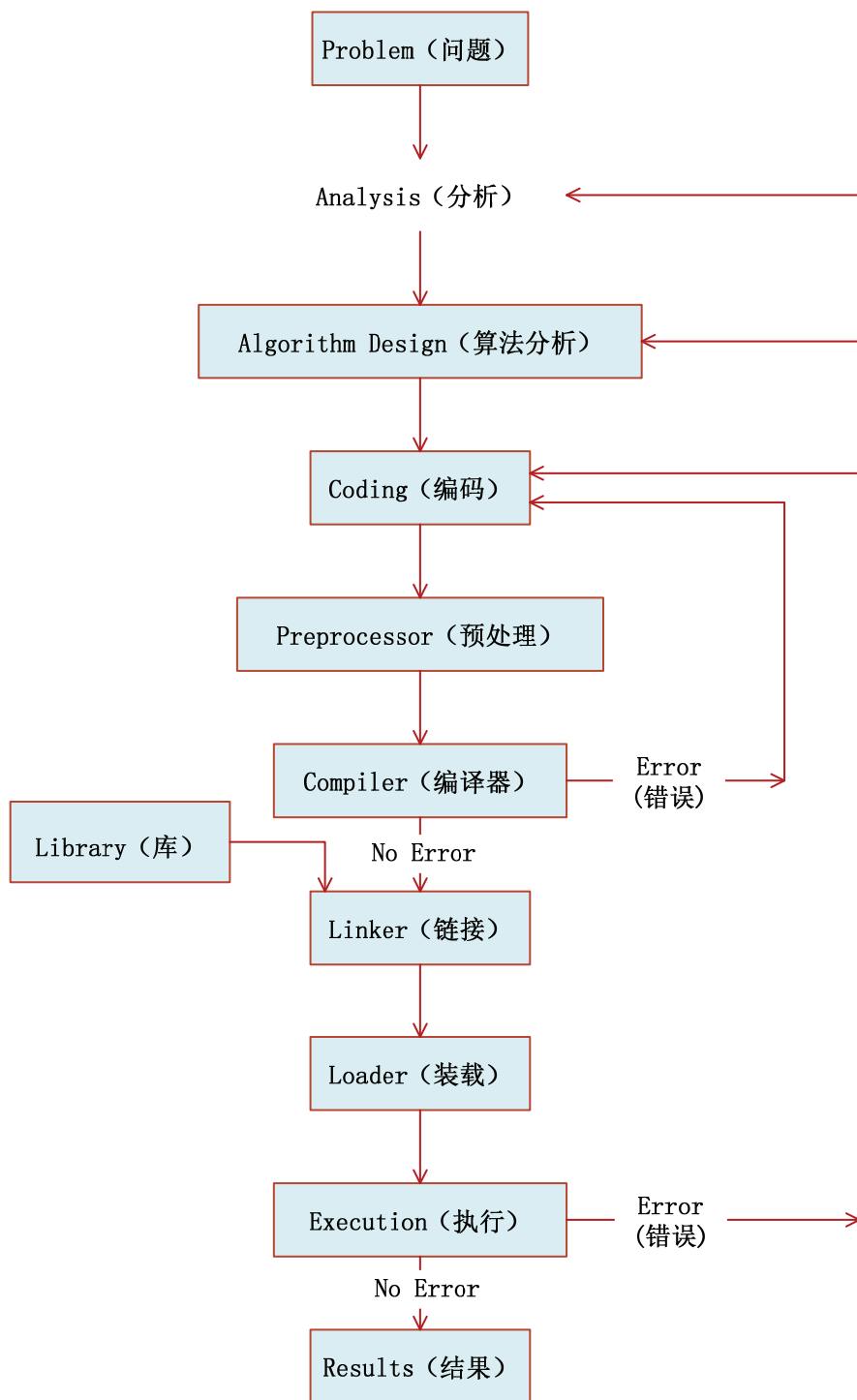


图 1-18 C++解决问题的流程

使用 VC 进行 C++程序设计时，程序员就可以把工作集中到问题分析、算法优化和编码上，其他工作就由 VC 来完成，从而大大减小了程序员的工作量。

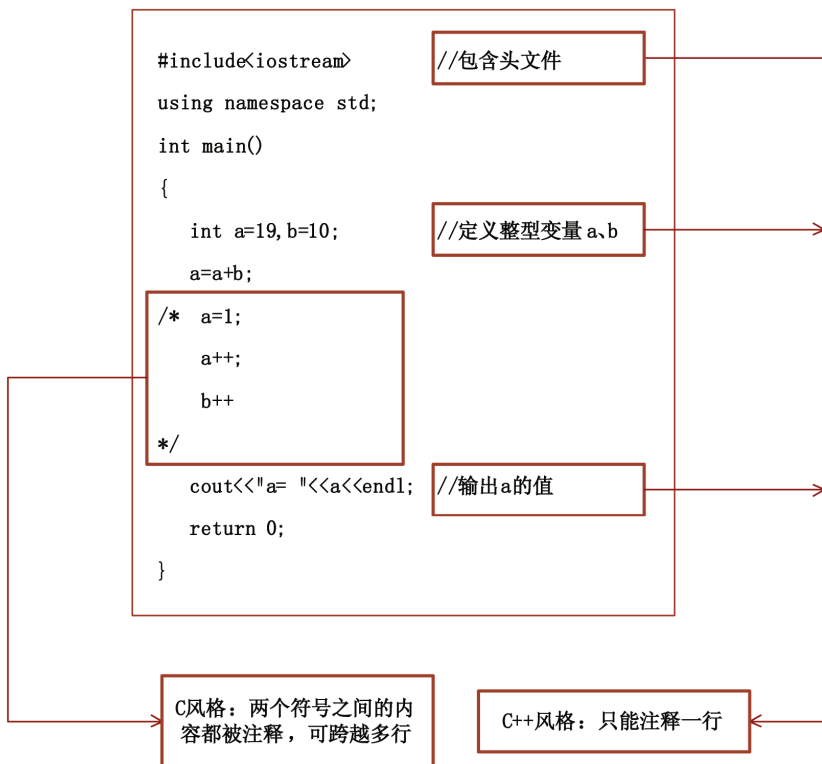


## 1.4 C++程序的结构

每个 C++ 程序都由注释、编译预处理和程序主体 3 部分组成，下面分别进行讨论。

### 1.4.1 注释

注释是程序员为读者提供的说明，是提高程序可读性的一种手段。注释仅供他人阅读程序时使用，是程序的可选部分，C++ 编译器忽略所有的注释，将其视为空白。C++ 支持两种风格的注释，一种是 C++ 风格，另一种是 C 风格，如图 1-19 所示的程序。



**说明：**应当养成在编码的同时随手写注释的好习惯，程序越复杂，就越有必要写注释，注释不仅方便他人阅读代码，也有助于程序员的总结、检查和回顾。

### 1.4.2 编译预处理和旧标准

以符号#开头的行，称为编译预处理行，如图 1-10 所示的代码使用了#include 编译指令，其作用是在编译之前将 iostream 文件的内容添加到程序中，iostream 设置了 C++ 的输入/输出环境，如 cin 和 cout 是在 iostream 中定义的 C++ 标准输入/输出设备标识符，endl 是 iostream 中定义的换行符。

前面已经说过，C++ 是 C 语言的超集，早期的 C 语言和 C++ 的头文件都采用扩展名为.h 形式，随着 C++ 的发展，关于头文件的标准也在不断变化，图 1-20 给出了新旧标准的对比。

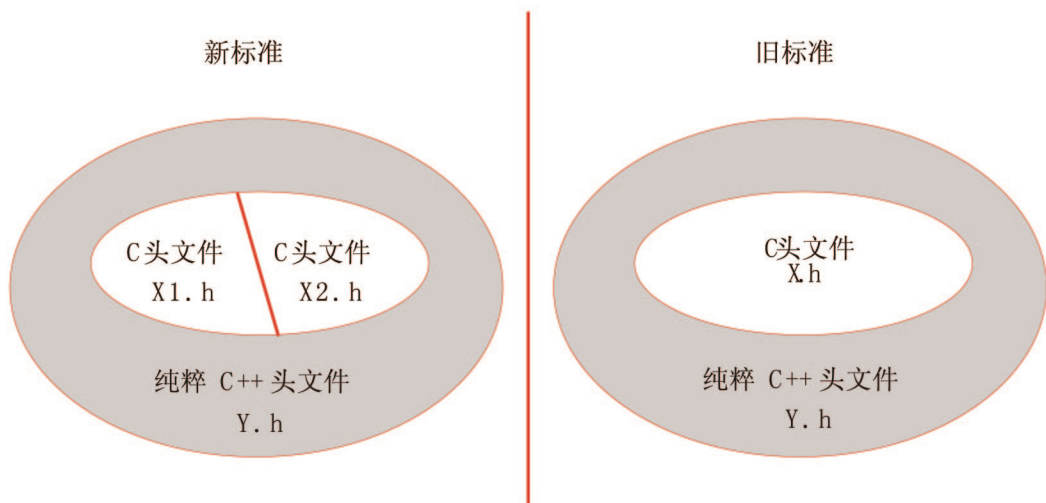


图 1-20 头文件新旧标准变化对比



**注意：**使用 `cin` 和 `cout` 进行输入和输出时，程序中必须包含 `iostream` 文件（对旧式编译器，程序中应包含 `iostream.h` 文件）。

### 1.4.3 程序主体

图 1-10 所示的程序中只有一个主函数。当然，一个程序中还可以有很多普通函数。主函数会自动被启动代码调用。启动代码是在编译阶段由编译器添加到可执行文件中的，是程序与操作系统的桥梁。因此，主函数是 C++ 程序的入口，每个 C++ 程序有且仅有一个主函数。`main` 函数在文件中的位置并没有特别的要求，它可以在文件的头部、中部或尾部。主函数可以带参数，也可以不带参数。关于函数的其他内容在后续章节中将会详细介绍。



## 1.5 小结

本章简要介绍了 C++ 的发展、特点，重点讲解了本书使用到的 C++ 编译器——Visual C++ 6.0 的集成开发环境，并在该环境中编写了第一个 C++ 源程序，详细介绍了在其中建立源程序、编译链接、调试、运行等步骤。最后，又对 C++ 源程序的组成结构进行了讲解。



## 1.6 习题

**【题目 1-1】**在 VC 中创建一个文件名为 `ex1.cpp` 的文件。完成后编写代码，在屏幕上输出一行字符 “I Love C++.”，实现效果如图 1-21 所示。

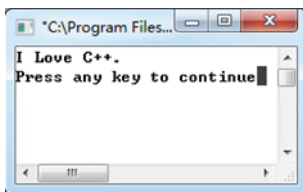


图 1-21 题目 1 的执行效果

【题目分析】 本题主要考查在 VC 中创建源文件的方法及简单程序的编写。

【关键代码】 以下是关键代码：

```
cout<<"I Love C++."<<endl;
```

下面给出本题的完整代码，在以后的习题中将不再给出完整代码。

```
#include <iostream>
using namespace std;
int main()
{
    cout<<"I Love C++."<<endl;
    return 0;
}
```

## 第 2 篇 面向对象篇

# 第2章 数据的表示

程序的主要目的是进行各种数据处理。计算机的数据多种多样，如数、文字、符号、图形、图像、声音等。了解数据在计算机中的表示方法，有助于读者更好地理解程序。本章将讲解数在计算机中的表示形式、数据的基本类型及变量的使用。

## 2.1 数

数是最简单的数据形式，数也是计算机直接能表示的。下面将讲解常见的数的表示形式，二进制、八进制、十六进制的构成与转换。

### 2.1.1 二进制

二进制是计算机的内存储器上唯一能识别的编码，其基本符号是“0”和“1”。

#### 1. 二进制的构成

二进制数据是用 0 和 1 两个数码来表示的数。它的基数为 2，进位规则是“逢二进一”，借位规则是“借一当二”。二进制的具体表示形式如表 2-1 所示。

表 2-1 二进制的表现形式

十进制	二进制
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001

#### 2. 十进制转换为二进制——辗除法

表 2-1 只给出一个基本的十进制和二进制的对应关系。要想获得更多转换关系，可以使用辗除法。辗除法也就是“除模取余”法。下面以十进制的“19”转换为二进制为例进行讲解，如图 2-1 所示。十进制“19”转换成二进制的形式为“10011”。



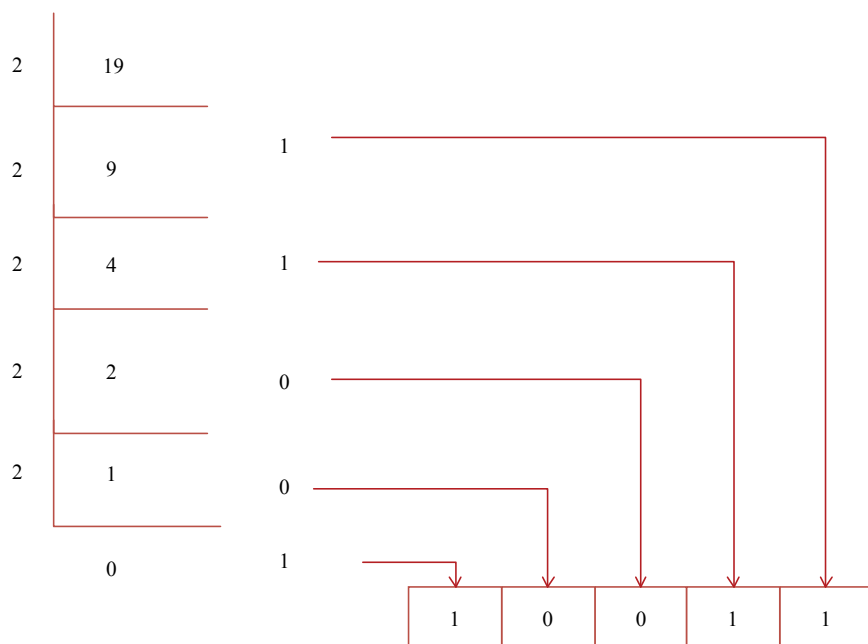


图 2.1 十进制向二进制的转换

## 2.1.2 八进制

由于二进制表示数字比较复杂，人们逐步使用八进制进行数据表示。下面讲解八进制的构成及其与二进制的转换。

### 1. 八进制的构成和表示

八进制是以数字 0 开头，用 0、1、2、3、4、5、6、7 八个数表示的。它的基数为 8，进位规则是“逢八进一”。八进制的表示如表 2-2 所示：

表 2-2 八进制的表示形式

十进制	二进制	八进制
0	0	00
1	1	01
2	10	02
3	11	03
4	100	04
5	101	05
6	110	06
7	111	07
8	1000	010
9	1001	011

### 2. 二进制和八进制的转换

二进制向八进制的转换是每三位二进制数转换为一位八进制数，运算的顺序是从低位向高位依次进行。以二进制“1111”为例，如图 2-2 所示。

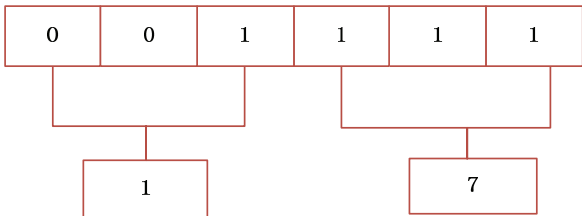


图 2-2 二进制向八进制转换

八进制向二进制转换的思路是八进制的一位转换为二进制的三位，运算的顺序是从低位向高位依次进行。以八进制“17”为例，具体转换如图 2-3 所示。

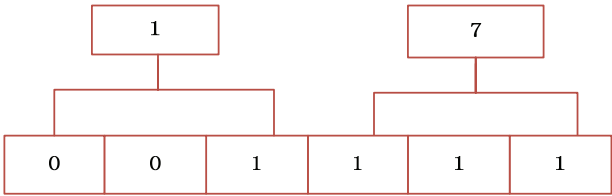


图 2-3 八进制转换为二进制

2.1.3 十六进制

虽然有了八进制，但是遇到更大的数字，八进制表示还是不方便，于是十六进制就诞生了。下面讲解十六进制的构成及其与二进制的转换。

1. 十六进制的构成和表示

十六进制必须以数字 0 和字母 x 开头（即 0x），是用 0~9 及 A、B、C、D、E、F 十六个数表示的。它的基数为 16，进位规则是“逢十六进一”。十六进制的表示如表 2-3 所示。

表 2-3 十六进制的表示形式

十进制	十六进制
0	0x0
1	0x1
2	0x2
3	0x3
4	0x4
5	0x5
6	0x6
7	0x7
8	0x8
9	0x9
10	0xA
11	0xB
12	0xC
13	0xD
14	0xE
15	0xF
16	0x10
17	0x11



## 2. 二进制和十六进制的转换

二进制向十六进制转换时，四位转换成十六进制的一位，运算的顺序是从低位向高位依次进行。以“1110011”转换成十六进制为例，具体实现如图 2-4 所示。

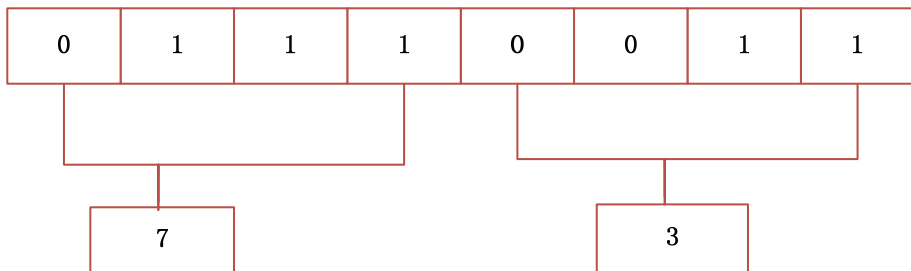


图 2-4 二进制向十六进制转换



**注意：**从低位向高位进行，如果最后不够四位，则在缺少的高位补 0。

十六进制向二进制转换，就是把十六进制的一位转换成二进制的四位，注意，运算的顺序是从低位向高位依次进行。以十六进制“73”为例，具体转换如图 2-5 所示。

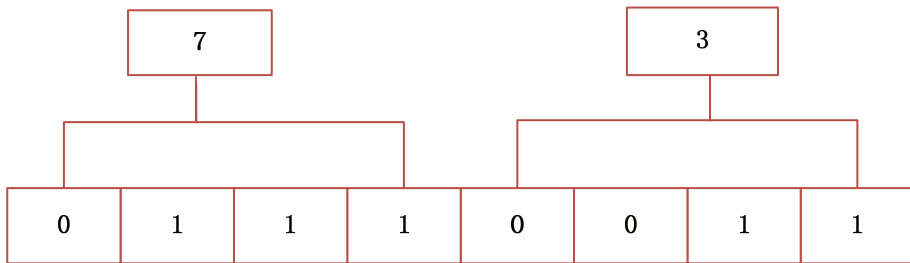


图 2-5 十六进制向二进制转换



## 2.2 数据的描述

计算机能够直接表示的数据只有数一种，但常用的数据类型有多种。为了表示这些不同类型的数据，C++语言提供了多种转换方式。本节将讲解 C++ 构词方式及几种基本数据类型。

### 2.2.1 C++的构词方式——标识符

C++语言作为一种语言，也像汉语一样，有一定的构词方式。按照规定的构词方式构成的词语，称为标识符。标识符的构成规则如图 2-6 所示。



长度1…32位，该长度是由机器的编译系统决定的，本书采用的是32位

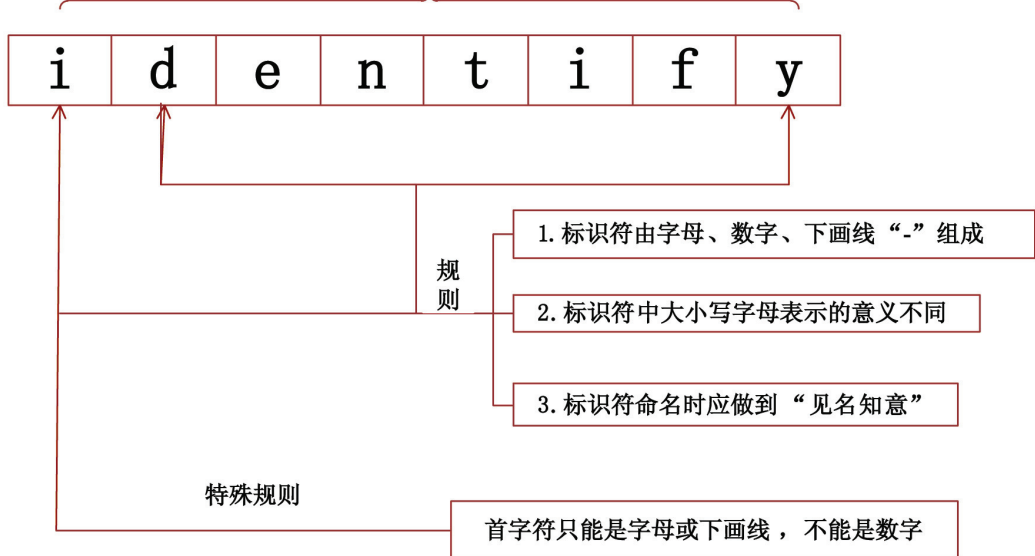


图 2-6 标识符

下面的标识符名是合法的：

`year, Day, ATOK, x1, _CWS, _change_to`

而图 2-7 所示的标识符名是不合法的。

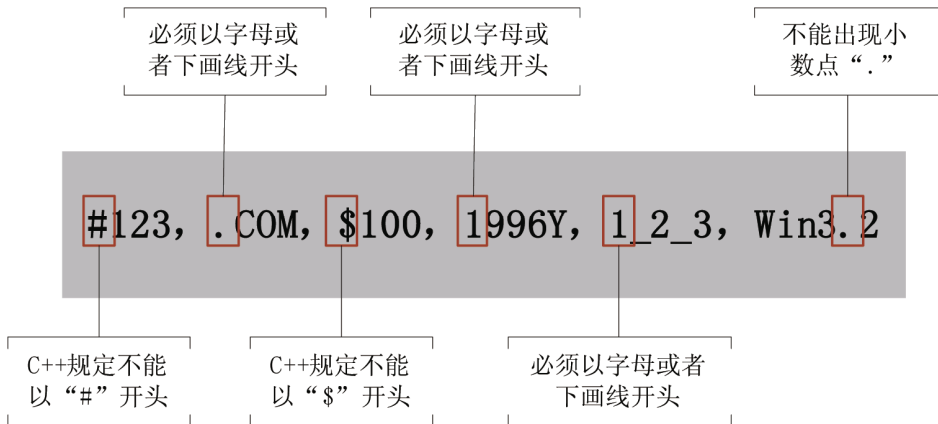


图 2-7 不合法的标识符

C++中的标识符是区分大小写的，例如，“Abcd”与“abad”是两个不同的标识符。

## 2.2.2 特殊的标识符——关键字

关键字又称保留关键字，是指被系统已经用于特定用途的标识符。这些标识符不能再被用户使用。C++语言常见的关键字如表 2-4 所示。



表 2-4 C++常用关键字

asm	auto	bool	break	case	catch	char
class	const	continue	default	delete	do	double
else	enum	extern	false	float	for	friend
goto	if	inline	int	long	main	namespace
new	operator	private	protected	public	register	return
short	signed	sizeof	static	struct	switch	template
this	throw	true	try	typedef	typeid	typename
union	unsigned	using	virtual	void	volatile	while

### 2.2.3 整数类型

整数类型是用来表示一个整数的。在 C++ 语言中，使用关键字 `int` 表示。由于计算机存储字长的限制，整型所能表示的数的范围是数学概念下的子集。整型数可用十进制、八进制、十六进制三种进制表示。

#### 1. 整数类型的二进制数表示

整数是以二进制数形式在计算机中存储的。存储时分为有符号型和无符号型。下面以整数 45 为例，它的二进制数存储形式如图 2-8 所示。

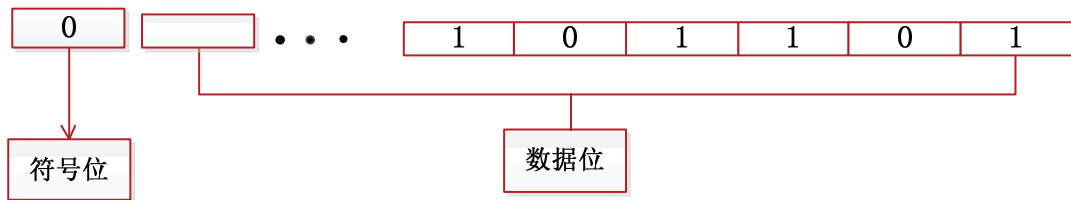


图 2-8 整数在计算机中的存储

#### 2. 扩展整数类型——short、long

`short` 类型说明符为 `short int` 或 `short`，其取值为短整型。`long` 类型说明符为 `long int` 或 `long`，其取值为长整型。这两种类型所占内存字节及表示数的范围如表 2-5 所示。

表 2-5 short和long型

类型说明符	内存字节	数的范围
Short	2	-32 768~32 767
Long	4	-2 147 483 648~2 147 483 647



**注意：**本书以 Visual C++ 6.0 32 位编译器为准。

#### 3. 无符号类型——unsigned

无符号类型的类型说明符为 `unsigned`。它可以单独使用代表 `unsigned int`，也可以作为前缀，都表示无符号整数，即永远为非负的整数。下面以无符号型整数 45 为例，从图 2-9 中可以看出无符号型在内存中的存储形式。

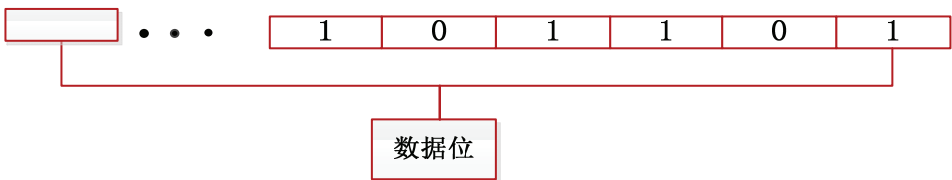


图 2-9 无符号型的二进制存储形式

4. 总结

C++支持的整数类型共有 6 种，如表 2-6 所示。

表 2-6 6 种整数类型

类型说明符	内存字节	数的范围
short	2	−32 768~32 767
int	4	−2 147 483 648~2 147 483 647
long	4	−2 147 483 648~2 147 483 647
undigned short	2	−32 768~32 767
undigned int	4	−32 768~32 767
undigned long	4	−2 147 483 648~2 147 483 647

2.2.4 浮点类型

浮点型数据即实数，实数就是在数轴上一一对应的点。我们常用浮点表示法来表示实数，也就出现了浮点类型。浮点数是利用指数使小数点的位置可以根据需要而上下浮动，从而可以灵活地表达更大范围的实数。

1. 浮点类型的二进制表示

浮点数真值的二进制表达式是： $N=\pm 2E \cdot M$ 。表达式的解释如图 2-10 所示。

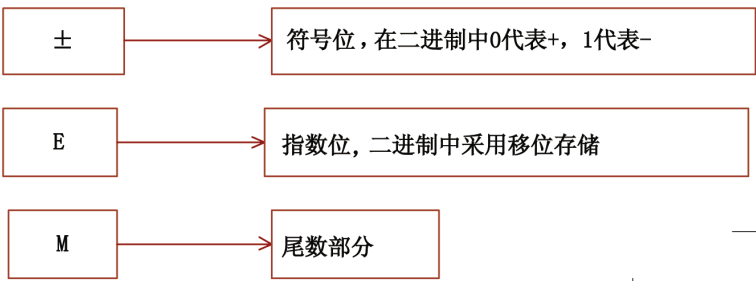


图 2-10 浮点数表达式

二进制存储形式如图 2-11 所示。



图 2-11 浮点类型的二进制表示



## 2. 扩展浮点类型——float、double

根据浮点的精度不同，浮点类型分为单精度类型（float）和双精度类型（double）。根据 VC++ 6.0 32 位编译器，float 占 4 字节，double 占 8 字节。float 和 double 类型在二进制中存储的符号位、指数位及尾数位所占的位数不同，具体如表 2-7 所示。

表 2-7 float和double的对比图

浮点类型	符号位	指数位	尾数位
单精度（float）	1bit	8bit	23bit
双精度（double）	1bit	11bit	52bit

## 3. 总结

C++语言支持的浮点类型共有 3 种，如表 2-8 所示。

表 2-8 3 种浮点类型

类型	符号位	指数位	尾数位	有效数字	数值范围
float	1bit	8bit	23bit	6~7bit	$3.4 \times 10^{-38} \sim 3.4 \times 10^{38}$
double	1bit	11bit	52bit	15~16bit	$1.7 \times 10^{-308} \sim 1.7 \times 10^{308}$
long double	1bit	11bit	52bit	15~16bit	$1.7 \times 10^{-308} \sim 1.7 \times 10^{308}$

## 2.2.5 字符类型

C 语言中，每个字符在内存中占一字节，字符在内存中都是以 0 和 1 代码存储的，并且存储的值是其与 ASCII 码表对应的 ASCII 码值。ASCII 码表如表 2-9 所示。

表 2-9 ASCII 码表

ASCII 值	控制字符	ASCII 值	控制字符
0	NUT	21	@
1	SOH	22	A
2	STX	23	B
3	ETX	24	C
4	EOT	25	D
5	ENQ	26	E
6	ACK	27	F
7	BEL	28	G
8	BS	29	H
9	HT	30	I
10	LF	31	J
11	VT	32	K
12	FF	33	L
13	CR	34	M
14	SO	35	N
15	SI	36	O
16	DLE	37	P
17	DC1	38	Q
18	DC2	39	R
19	DC3	40	X
20	DC4	41	T



续表

ASCII 值	控制字符	ASCII 值	控制字符
42	*	85	U
43	+	86	V
44	,	87	W
45	-	88	X
46	.	89	Y
47	/	90	Z
48	0	91	[
49	1	92	\
50	2	93	]
51	3	94	^
52	4	95	_
53	5	96	`
54	6	97	a
55	7	98	b
56	8	99	c
57	9	100	d
58	:	101	e
59	;	102	f
60	<	103	g
61	=	104	h
62	>	105	i
63	?	106	j
64	@	107	k
65	A	108	l
66	B	109	m
67	C	110	n
68	D	111	o
69	E	112	p
70	F	113	q
71	G	114	r
72	H	115	s
73	I	116	t
74	J	117	u
75	K	118	v
76	L	119	w
77	M	120	x
78	N	121	y
79	O	122	z
80	P	123	{
81	Q	124	
82	R	125	}
83	X	126	~
84	T	127	DEL

表 2-9 中给出了控制字符，下面在表 2-10 中进行简单说明。





表 2-10 控制字符说明表

NUL 空	VT 垂直制表	SYN 空转同步
SOH 标题开始	FF 走纸控制	ETB 信息组传送结束
STX 正文开始	CR 回车	CAN 作废
ETX 正文结束	SO 移位输出	EM 纸尽
EOY 传输结束	SI 移位输入	SUB 换置
ENQ 询问字符	DLE 空格	ESC 换码
ACK 承认	DC1 设备控制 1	FS 文字分隔符
BEL 报警	DC2 设备控制 2	GS 组分隔符
BS 退一格	DC3 设备控制 3	RS 记录分隔符
HT 横向列表	DC4 设备控制 4	US 单元分隔符
LF 换行	NAK 否定	DEL 删除

字符型数据类型只占据一字节。可以为其加上 `unsigned`、`singed` 修饰符，分别表示无符号字符型和有符号字符型。其分别对应的取值范围如表 2-11 所示。

表 2-11 字符数据类型

数据类型	类型描述	占字节数	取值范围
<code>char</code>	字符型	1	-128~127
<code>unsigned char</code>	无符号字符型	1	0~255
<code>signed char</code>	有符号字符型	1	-128~127

## 2.2.6 布尔类型

布尔型是最简单的数据类型，其只有两个值：`true` 和 `false`，占用一字节。把一个整型变量转换成布尔型变量时，其对应关系如图 2-12 所示。

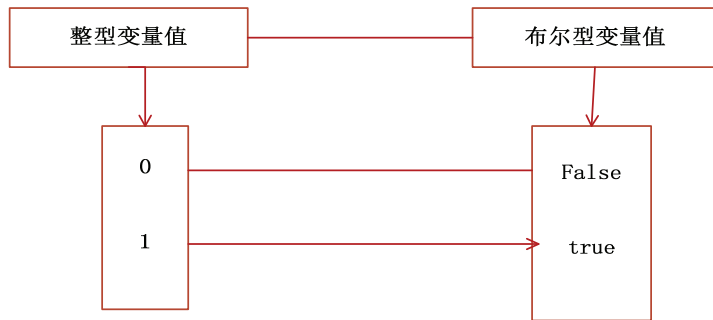


图 2-12 整型与布尔型转换



**说明：**非零整型转换为布尔型时，对应布尔型的 `true`。



## 2.3 变量

为了方便表示数据，在编程时往往使用一个特定的标识符指代特定的数据。该标识符称为变量。本节将讲解如何使用变量。



### 2.3.1 变量的声明和定义

变量在使用之前必须首先声明和定义。声明变量用来建立这个标识符。定义变量实现为该变量指代的数据分配内存空间。在 C++ 语言中，变量的声明和定义是同时实现的。变量的声明和定义语法如图 2-13 所示。

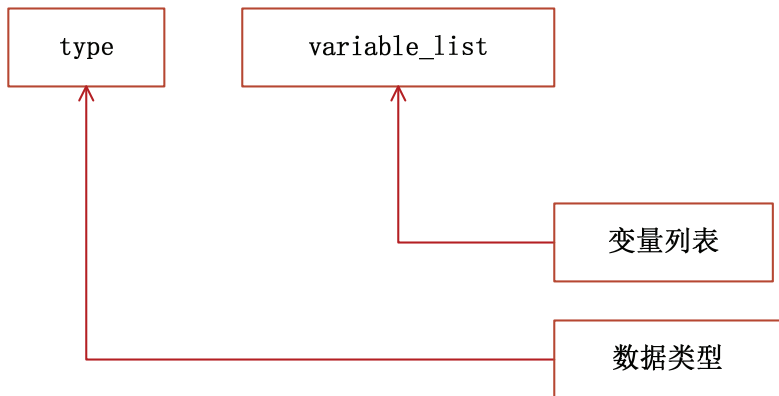


图 2-13 变量的声明和定义

【示例 2-1】下面声明和定义一个变量 `age`，用来指代人的年龄。

```
int age;
```

上述代码声明了一个标识符 `age`。在程序中，使用 `age` 指代数据年龄。在运行时，系统会为年龄数据分配一个 4 字节的内存空间。

### 2.3.2 不变的变量——常量

在程序中，有很多变量所代表的值并不会发生变化。为了防止这类变量被意外修改，直接将这类变量定义为常量。常量的定义方式如图 2-14 所示。

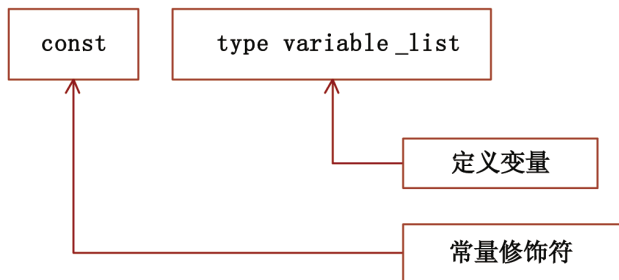


图 2-14 常量的定义方式

【示例 2-2】下面定义一个常量 `age`，用来指代人的年龄。

```
const int age;
```

上述代码声明了一个标识符 `age`。在程序中，使用 `age` 指代年龄。在运行时，年龄数据是不变的。



【示例 2-3】下面的例子将常数圆周率设置为常变量。

```
#include<iostream>

using namespace std;

int main()
{
    const double _PI_=3.1415926;

    double r;

    r=4.89;

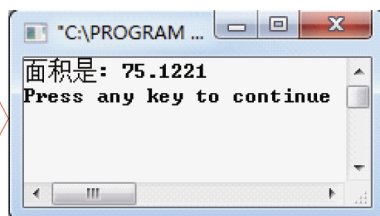
    double area;

    area=_PI_*r*r;

    cout<<"面积是："<<area<<endl;

    return 0;
}
```

运行  
结果



## 2.4 小结

本章主要讲解了数的几种进制、数据类型及常量变量，重点是掌握计算机系统最终是以二进制数的形式存储数据的，以及不同类型的数据在计算机中是如何存储的。

## 2.5 习题

【题目 2-1】将十进制数 30、八进制数 0216、十六进制数 0x5A6 转换成计算机中的存储表示。

【题目分析】要解答本题，读者首先应知道计算机只能直接识别二进制数据，并且以二进制来存储数据。了解了这一点，才能知道本题是将其他进制的数据转换为二进制类型。除此之外，还要明白十进制数转换为二进制数的方法是除二取余，将余数逆序排列得到的就是二进制数的表示形式。八进制数转换为二进制数的方法是八进制数的每一位分别转换为二进制数即可。十六进制数转换为二进制数的方法与八进制数类似。



【关键步骤】十进制数 30 转换为二进制数如图 2-15 所示。

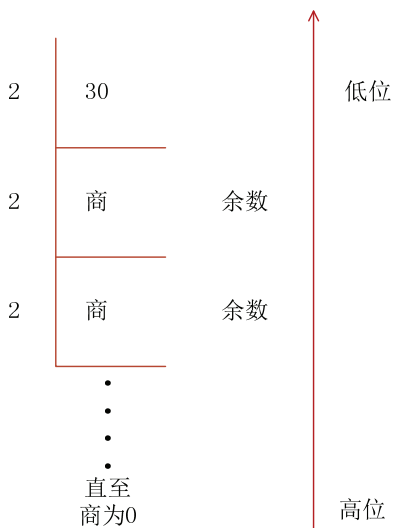


图 2-15 十进制数转换为二进制数

八进制数、十六进制数转换为二进制数如图 2-16 所示。

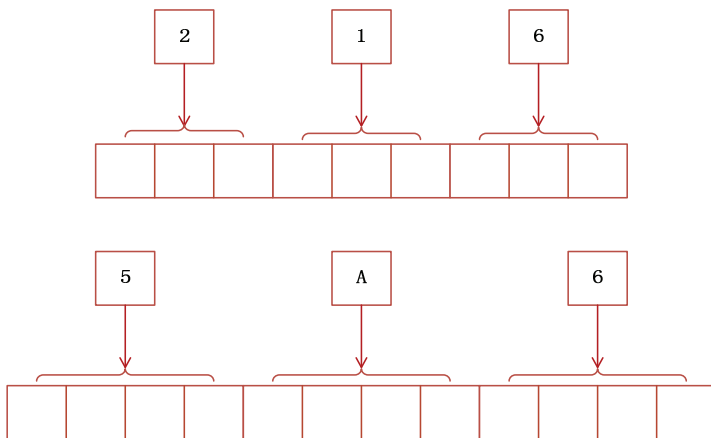


图 2-16 八进制数、十六进制数转换为二进制数

【题目 2-2】小明在 2000 年的年龄是 12 岁，身高为 1.50 米，在学校的成绩为 A 等。假设小明每年长高 5 厘米，那么小明在哪一年时年龄为 15 岁，身高为多少？并输出小明在 2000 年的成绩等级。要求：通过编程计算小明的年龄和身高，并将所得的结果输出到显示器上。程序的运行效果如图 2-17 所示。

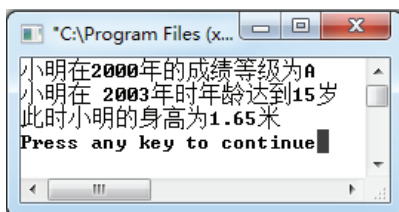


图 2-17 运行效果



【题目分析】本题主要考查数据类型的基础知识。在定义数据时，应将小明的身高定义为浮点型，将年龄和年份定义为整型，将成绩等级定义为字符型。学习数据类型知识，主要目的在于能将数据定义为合适的数据类型。

#### 【关键代码】

```
int age, year;
char score;
double high;
score='A';
age=12;
year=2000;
high=1.50;
year=year+15-age;
high=high+0.05*(15-age);
cout<<"小明在 2000 年的成绩等级为"<<score<<endl;
cout<<"小明在 "<<year<<"年时年龄达到 15 岁"<<endl;
cout<<"此时小明的身高为"<<high<<"米"<<endl;
```

【题目 2-3】下面将给出一段程序代码，这段代码在编译时由于标识不符合 C++ 规则而出现错误。请读者找出其中的错误，并运行出正确结果。代码如下：

```
#include<iostream>
using namespace std;
int main()
{
    const double PI=3.1415926;
    double r;
    r=4.89;
    double area;
    area=PI*r*r;
    cout<<"面积是: "<<Area<<endl;
    return 0;
}
```

错误信息如图 2-18 所示。

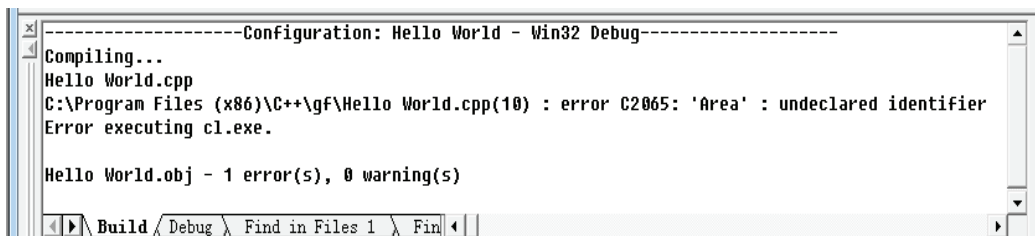


图 2-18 编译错误信息

运行的正确效果如图 2-19 所示。

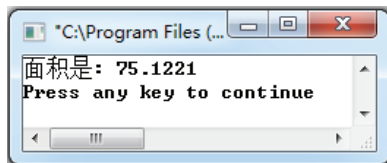


图 2-19 运行效果

【题目分析】本题主要考查标识符的命名规则，大小写表示不同的标识符。另外，标识符的第一个字母只能是字母或下划线。图 2-18 所示的错误提示信息指出“Area 变量未定义”。

**【关键代码】**

```
cout<<"面积是: "<<area<<endl;
```

**【题目 2-4】**编写代码计算球的体积和表面积。其中，圆周率定义为常量，半径、体积、表面积定义为单精度浮点类型的变量。下面给出表面积和体积的计算公式。

球的体积计算公式为： $v=3.14*r*r*r*4/3$ ;

球的表面积计算公式为： $s=4*3.14*r*r$ 。

程序的运行效果如图 2-20 所示。

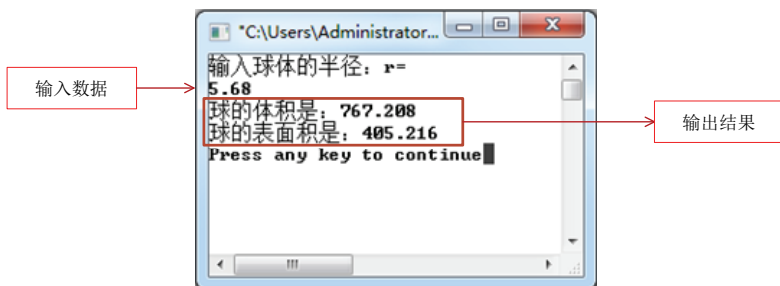


图 2-20 运行效果

**【题目分析】**本题主要考查常量的定义方法。

**【关键代码】**

```
const float PI=3.14;  
float v,s,r;  
cout<<"输入球体的半径: r= "<<endl;  
cin>>r;  
v=PI*r*r*r*4/3;  
s=4*PI*r*r;  
cout<<"球的体积是: "<<v<<endl;  
cout<<"球的表面积是: "<<s<<endl;  
return 0;
```

# 第 3 章 程序的基本单位——语句

语句和表达式是程序语言的基本组成部分。在程序中，数据处理是通过表达式和语句完成的。掌握语句、运算符是学习程序的基础。本章将讲解语句的构成、运算符和语句块。

## 3.1 语句的构成

在 C++ 中，语句是数据处理过程中的最小单位。C++ 中的基本语句有表达式语句、空语句、输入/输出语句、复合语句、控制语句、函数调用语句。本节主要讲表达式语句、输出/输入语句，其他语句将在后面的章节进行介绍。

### 3.1.1 表达式语句

表达式语句是由表达式和“;”构成的，而表达式是由变量和运算符构成的。表达式语句的构成如图 3-1 所示。

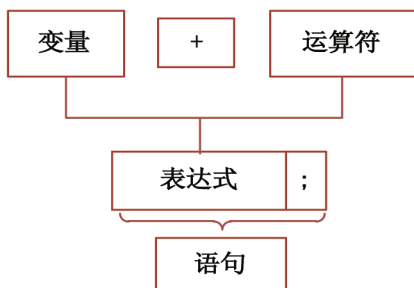


图 3-1 表达式语句的构成

此外，语句也可以是一个分号，这种语句称为空语句。空语句仅由一个分号组成，不进行任何操作。一般用于语法上要求有一条语句但实际没有任何操作的场合，如 for 语句中。

### 3.1.2 输入/输出语句

C++ 没有提供输入/输出语句，其输入/输出功能由函数（scanf()、printf()）或流控制来实现。输入/输出流（I/O 流）是输入或输出的一系列字节。C++ 定义了包含重载运算符“<<”和“>>”的 `iostream` 类。在这里只介绍如何利用 C++ 的标准输入/输出流实现数据的输入/输出功能。C++ 的标准输入/输出流如图 3-2 所示。

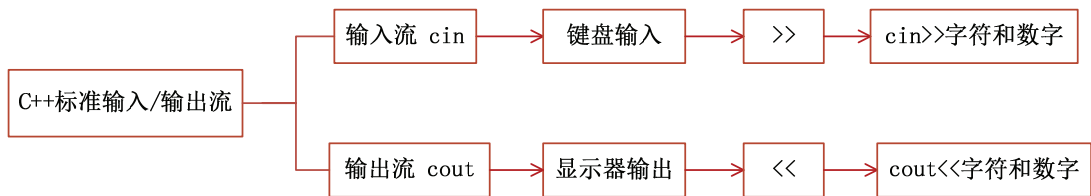


图 3-2 C++的标准输入/输出流

(1) 输出流 **cout**: 输出流的任务是将数值类型转换为以字符字节为单位的输出，这些是通过 **ostream** 类的公共方法（接口）来实现的。

(2) 输入流 **cin**: 输入是相对程序而言的，即如何从键盘和外部提供数据，头文件 **iostream** 中定义了 **istream** 流类库对象 **cin**。

### 1. 输出语句

当程序需要在屏幕上显示输出时，可以使用操作符“<<”向输出流 **cout** 中插入字符和数字，并把它在屏幕上显示输出。

【示例 3-1】该示例给出了最简单的输出语句的实现，其功能是输出字符串“Hello”，使其显示到屏幕上，实现代码及结果如图 3-3 所示。

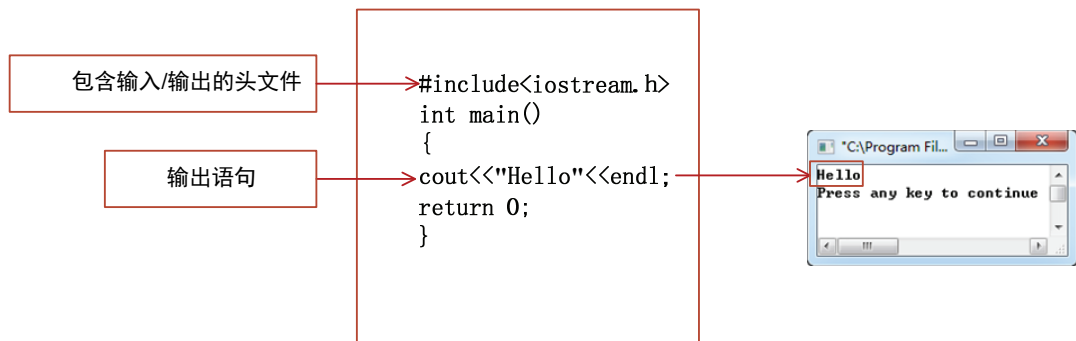


图 3-3 输出语句

### 2. 输入语句

当程序需要执行键盘输入时，可以使用操作符“>>”从输入流 **cin** 中抽取键盘输入的字符和数字，并把它赋给指定的变量。

【示例 3-2】该示例给出了最简单的输入语句的实现。该程序段的功能是接收用户从键盘输入的一个整数，并将其存储到变量 **a** 中输出，实现代码及结果如图 3-4 所示。

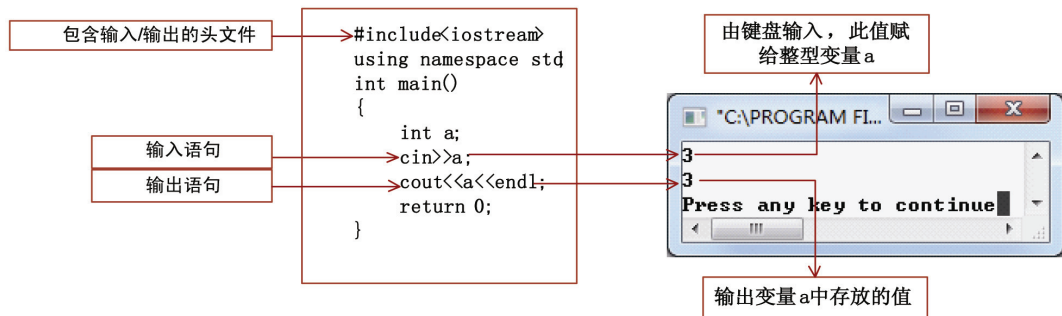


图 3-4 输入语句





第一个 6 是键盘输入的，此值传送给整型变量 `a`，第二个 6 是 “`cout<<a<<endl;`”，将 `a` 的值输出。



## 3.2 运算符

运算符是学习 C++ 程序的基础。本节简要介绍运算符的概念及分类。

### 3.2.1 运算符概述

C++ 中的运算符是可以让 C++ 编译器识别的具有运算意义的符号。编译器把这些符号及其组成的表达式翻译成相应的机器代码后，就可以由计算机运行得出正确的结果。例如，在日常生活中，冰箱、电视机等分别代表不同功能的电器设备，那么运算符就是代表 C++ 语言中的各个运算功能的名字，这些名字是由制定 C++ 语言规范的人员确定的，如图 3-5 所示。

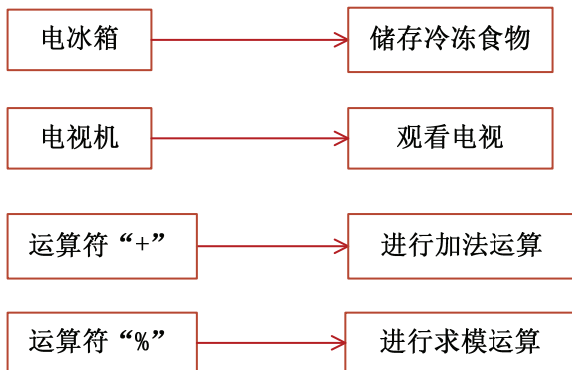


图 3-5 运算符的概念

C++ 中包含了 C 语言中几乎所有的运算符，并且在其基础上又增加了几种新的运算符，如图 3-6 所示。

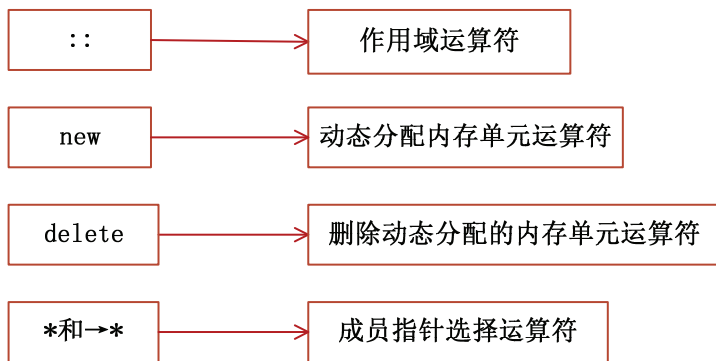


图 3-6 C++ 新的运算符

### 3.2.2 运算符的分类

根据运算符需要的操作数的个数，可将其分为 3 种，如图 3-7 所示。

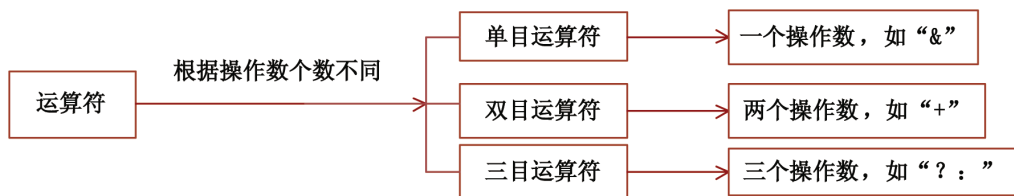


图 3-7 运算符根据操作数个数的分类



**说明：**单目运算符又称一元运算符。同理，双目运算符和三目运算符又称二元运算符和三元运算符。

在 C++ 中，三目运算符只有一个，就是条件运算符“?:”。

根据运算符实现的功能进行划分，C++ 提供的基本运算符有以下几种：赋值运算符、算术运算符、位运算符、关系运算符、逻辑运算符、条件运算符、逗号运算符、sizeof 运算符及其他运算符。下面将一一讲解这些运算符的功能和应用。



## 3.3 赋值运算符

赋值运算符是 C++ 基本运算符中最常用的。本节将介绍赋值运算符的使用和在赋值时的数据类型转换。

### 3.3.1 赋值运算符——“=”

赋值运算符是 C++ 程序设计中最基本的运算符之一。利用赋值运算符可以给一个变量赋值。C++ 的基本赋值运算符用“=”来表示，它是一个二元运算符，其一般形式如图 3-8 所示。

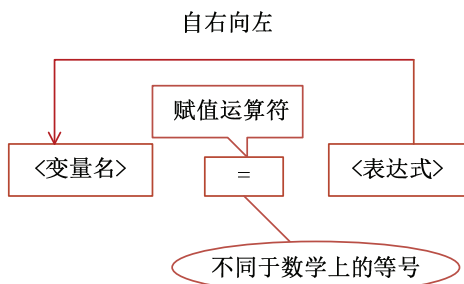


图 3-8 赋值运算符

例如，表达式 `a = b` 的含义是将右操作数 `b` 的值赋给左操作数 `a`，即用 `b` 的值覆盖 `a` 的值。在进行赋值时，需注意如下问题：

(1) 在程序中，以关键字 `const` 声明的符号常量进行赋值操作是错误的，因为常量是不允许改变其原有值的。

(2) 赋值运算符的结合性是自右向左的。例如，思考以下赋值语句的赋值顺序。

```
int x,y,z=4;  
x=y=z;
```

首先，变量 `z` 的值赋给 `y`，接下来把 `y` 的新值赋给 `x`。



### 3.3.2 数据类型转换——隐式转换

类型转换是用来把一个类型的值转换成另一个类型。当用户需要把一个值转换为另一个类型时，就需要使用一些方式进行类型转换。C++中，支持隐式转换和显式转换两种。本小节首先讲解隐式转换。

隐式转换是系统默认的，不需要加以声明就可以进行的转换。在隐式转换过程中，编译器无须对转换进行详细检查就能够安全地执行转换。比如从 `int` 类型转换到 `long` 类型就是一种隐式转换。隐式转换一般不会失败，转换过程中也不会导致信息丢失。

【示例 3-1】该示例实现数据类型的隐式转换，将字符型变量转换为整型变量，实现代码及其结果如图 3-9 所示。

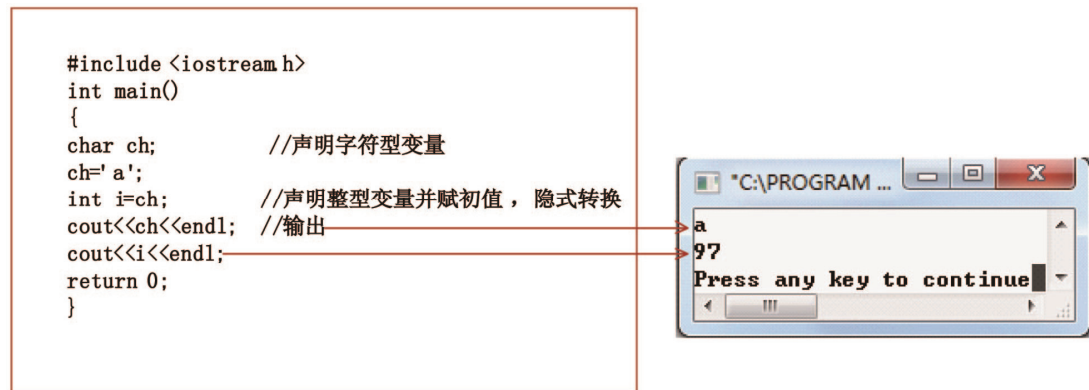


图 3-9 隐式转换实例

【示例 3-1】中声明了一个字符型变量 `ch` 和一个整型变量 `i`。并在声明变量 `i` 的同时给其赋初值 `ch` 的值，此处就需要将字符型的值转换为整型值，Visual C++ 编译器自动实现了它们之间的转换，称为隐式转换。



**提示：**图 3-9 中输出整型变量 `i` 的值为 97，这是因为字母 `a` 对应的 ASCII 码值为 97。由此可看出，字符型和整型数据类型之间可以相互转换。

### 3.3.3 显式转换

通常，隐式转换意味着编译器认为转换是合理的或者是安全的。因此，隐式转换可能出现意想不到的错误。为了避免隐式转换，C++ 还支持显式转换，显式转换是用户手动指出需要转换的类型。显式转换意味着编译器能够找到一个转换方式，但是它不保证这个转换是否安全，所以需要程序员额外指出。显示转换有多种方式，下面依次讲解几种常用的方式。

#### 1. 旧式转换

旧式转换是类似于 C 语言的一种转换方式。直接使用数据类型标识符来修饰要转化的变量。语法形式如下：

**(type) expression**

其中，`type` 为转换后的数据类型名称；`expression` 为要转换的变量或者表达式。



【示例 3-2】该示例实现了数据类型之间的显式转换，将字符型变量通过（）符显式转换为整型变量，实现代码及其结果如图 3-10 所示。

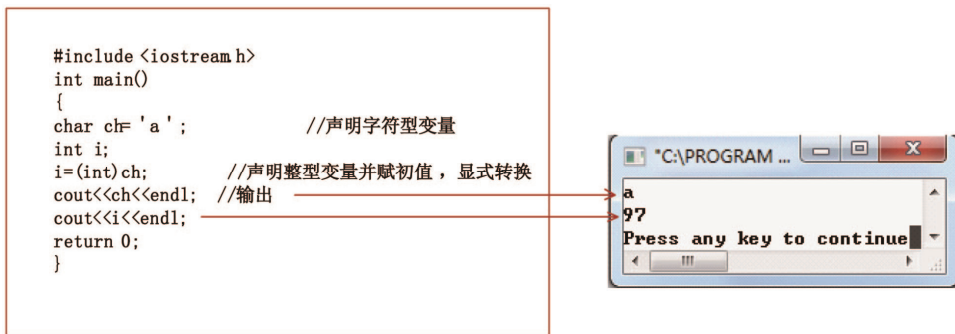


图 3-10 显式转换实例

【示例 3-2】中使用了语句“i=(int)ch;”将字符型变量 ch 的类型强制转换为整型，因此，其运行后的效果与图 3-9 相同。

## 2. 静态的转换方式

静态的转换方式发生在编译时，它可以显式完成编译器隐式执行的任何类型转换，屏蔽潜在精度损失产生的编译器警告。但不能用于指针类型转换。语法形式如下：

**static\_cast<type>(expression);**

其中，static\_cast 是修饰符，type 是转换后的数据类型名称，expression 为要转换的变量或者表达式。

【示例 3-3】图 3-11 中的代码将单精度数 a 显式转换为整型变量，并赋给整型变量 b。

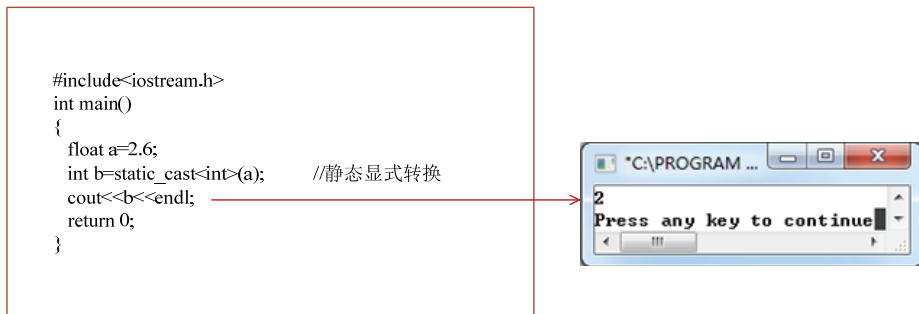


图 3-11 静态转换方式

单精度数转换为整型是取其整数部分，将小数部分直接丢弃。所以，2.6 转换为整型是 2。

## 3. 动态的转换方式

动态的转换方式提供了运行时确定对象类型的方法，支持运行时识别指针或引用所指向的对象。格式如下：

**dynamic\_cast<type>(expression);**

此方式用处很小，这里就不详说了。

## 4. 重新解释的转换方式

重新解释的转换方式常用于函数指针类型之间进行的转换，能够在非相关的类型之间转换。其操作结果只是简单的从一个指针到别的指针值的二进制复制，仅仅是重新解释了对象的



内存表示，对类型之间指向的内容不做任何类型检查和转换。格式如下：

```
reinterpret_cast<type>(expression);
```

【示例 3-4】 将一个字符类型的指针重新解释为整型类型的指针，如图 3-12 所示。

```
#include<iostream.h>
int main()
{
    char a[]="abc";           //定义字符数组变量
    char *p1=a;               //定义指针变量
    short int *p2=reinterpret_cast<short int*>(p1); //强制类型转换
    cout<<a[0]<<a[1]<<endl;
    cout<<p1<<endl;
    cout<<*p2<<endl;
    return 0;
}
```

图 3-12 重新解释转换方式

主要看第三行的输出结果，短整型指针变量 p2 是被字符指针变量 p1 初始化的。代码“reinterpret\_cast<short int\*>(p1)”将字符型指针 p1 重新解释为短整型指针变量赋给 p2。这样 p2 实际上指向 a[0]和 a[1]两个变量所占的两字节内存，所以输出为 25185。

5. 常量的类型转换

常量的类型转换用来转换掉对象的 const 或 volatileness 属性。任何修改 const 或 volatileness 之外的属性的企图都被拒绝。格式如下：

```
const_cast<type>(expression);
```

这种用法读者了解即可。

3.4 其他常用运算符

C++中除了赋值运算符外，还有一些常用的运算符，例如，算术运算符、自增自减运算符、关系运算符、逻辑运算符、位运算符、复合赋值运算符、条件运算符、逗号运算符、sizeof 运算符等。本节将详细介绍几种运算符的使用及其优先级。

3.4.1 算术运算符

算术运算符包括加、减、乘、除运算符 (+、-、\*、/) 和求模运算符 (%)。算术运算符是双目运算符，其操作数一般是整数和浮点数（或者是结果为整数或浮点数的表达式）。C++语言中支持的算术运算符符号、名称、功能及其相关示例如表 3-1 所示（已知 a=10，b=4）。

表 3-1 算术运算符

运 算 符	运算符名称	功 能	示 例	结 果
+	加法运算符	表示两个数相加	a+b	14
-	减法运算符	表示两个数相减	a-b	6
*	乘法运算符	表示两个数相乘	a*b	40
/	除法运算符	表示两个数相除	a/b	2.5
%	模运算符	表示取模	a%b	2



**提示：**在算术运算符中，如果两个操作数都是整数，则运算的结果也是整数。

在进行包含多个算术运算符的表达式运算时，需要注意算术运算符的优先级和结合性。算术运算符的结合性都是从左到右；算术运算符的优先级如图 3-13 所示。



图 3-13 算术运算符的优先级

【示例 3-3】已知 a=4, b=3, c=5, d=7, e=8, 求表达式 a+b-c/d+ e%d 的值，实现代码及结果如图 3-14 所示。

```
#include <iostream.h>    //预处理文件
int main()              //主函数
{
    int a=5;
    int b=3;
    int c=1, d=2, e=6;   //定义操作数变量并赋初值
    int res=0;           //定义存放运算结果的变量
    res=a+b-c/d+e%d;     //将运算结果存放在变量 res中
    cout<<"运算结果为: " <<res<<endl;
    return 0;
}
```

图 3-14 算术运算符实例



**注意：**C++中的除法运算与算术中的除法运算一样，除数都不可为 0，否则会导致程序崩溃。

### 3.4.2 自增自减运算符

自增、自减运算符“++”和“--”也可包含在算术运算符的范畴中。它们的表示形式和实现功能如图 3-15 所示。

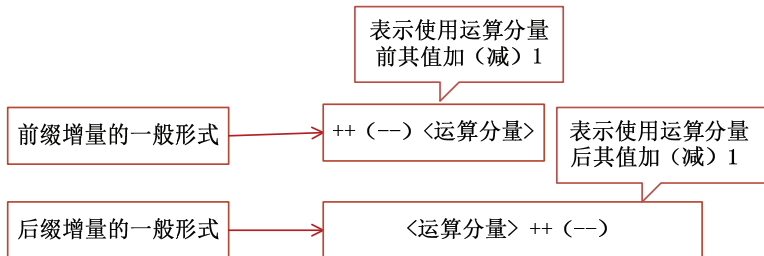


图 3-15 自增自减运算符的表现形式和功能



自增自减运算符其实是简化变量加（减）1 的操作，如图 3-16 所示。



图 3-16 变量加（减）1 的简化

自增、自减运算符“++”和“--”，其前缀增量和后缀增量所产生的结果是不同的。前增量表示先使用操作数的值，再对操作数作自增自减运算，后增量则表示先对操作数进行自增自减操作，再使用操作数。



**注意：**自增自减运算符的操作数必须是可修改的变量，而不能是常量。

【示例 3-4】分别使用前缀和后缀两种形式，仔细观察变量值的变化，其实现代码和输出结果如图 3-17 所示。

```
#include <iostream.h>
int main()
{
    int a=6;
    int b=a++;          //该条语句执行后，b的值为6，a的值为7
    int c=a--;          //该条语句执行后，c的值为7，a的值为6
    cout<<"a="<<a<<" "<<"b="<<b<<" "<<"c="<<c<<endl;
    int d=6;
    int e=++d;          //该条语句执行后，e的值为7，d的值为7
    int f=--d;          //该条语句执行后，f的值为6，d的值为6
    cout<<"d="<<d<<" "<<"e="<<e<<" "<<"f="<<f<<endl;
    return 0;
}
```

图 3-17 自增自减运算符实例

3.4.3 位运算符

C++提供了对数据进行位运算的功能，包含 6 种位运算符，如表 3-2 所示。

表 3-2 位运算符

运算符	运算符名称	功能	实例	结果
&	按位与	表示 a 与 b 按位与	二进制 1001&0101	二进制 0001
	按位或	表示 a 与 b 按位或	二进制 1001 0101	二进制 1101
^	按位异或	表示 a 与 b 按位异或	二进制 1001^0101	二进制 1100
>>	右移位	表示 a 右移 b 位	二进制 1001>>2	二进制 0010
<<	左移位	表示 a 左移 b 位	二进制 1001<<1	二进制 0010
~	按位取反	表示 a 按位取反	二进制 ~1001	二进制 0110

&、|、^、~的运算规则如图 3-18 所示。

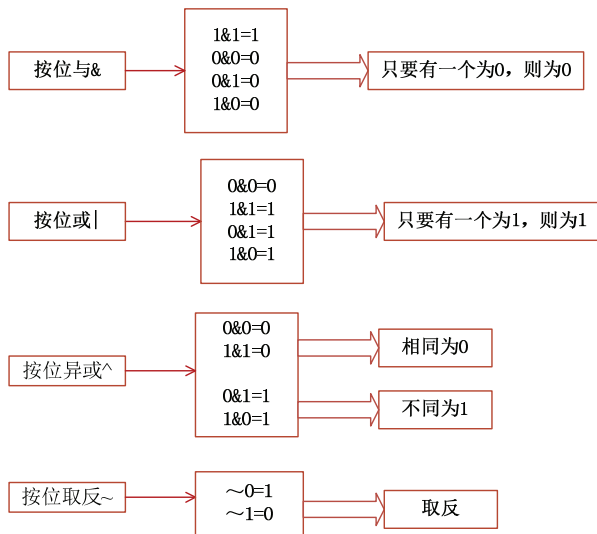


图 3-18 &、|、^、~的运算规则

左移位<<、右移位>>的运算规则如表 3-3 所示。

表 3-3 <<、>>的运算规则

符号	名称	作用	意义	说明
<<	左移位	将操作数的二进制位整体按照顺序向左移，右端空出的位补 0，左端移出最高位意外的位丢失	a<<n，代表 a 的二进制位顺序左移 n 位（n 为整数）	
>>	右移位	将操作数的二进制位整体按照顺序向右移，左端补 0 或补 1，右端移出超出最低位被丢失	a>>n，代表 a 的二进制位顺序右移 n 位（n 为整数）	右移运算的结果与操作数的符号有关，就 VC++ 6.0 而言：无符号数为“逻辑右移”，即左端空出位一律补 0；有符号数为“算术右移”，即正数右移，空位补 0，负数右移，空位补 1

位运算符的结合性除单目运算符按位取反是从右到左，其余双目位运算符都为从左到右。

【示例 3-7】下列程序对两个变量进行按位与、按位或、按位异或等几种位运算，将结果赋给变量并输出。其实现代码及结果如图 3-19 所示。

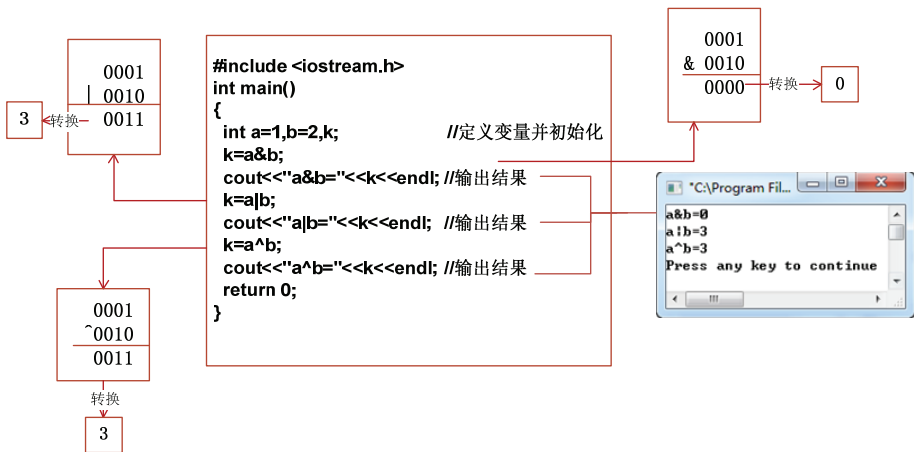


图 3-19 位运算符实例





3.4.4 复合赋值运算符

在程序中经常出现类似于 `s=s+n` 这样的赋值语句，C++允许采用更为简洁的形式写为“`s+=n`”，于是形成了复合赋值运算符。根据 C++中算术运算符和比较运算符的种类，复合赋值运算符一共有 10 种，如表 3-4 所示。

表 3-4 复合赋值运算符

运 算 符	使用方法	等效形式	说 明
<code>+=</code>	<code>a+=b</code>	<code>a=a+b</code>	将 a 加 b 的值赋给 a
<code>-=</code>	<code>a-=b</code>	<code>a=a-b</code>	将 a 减 b 的值赋给 a
<code>*=</code>	<code>a*=b</code>	<code>a=a*b</code>	将 a 乘以 b 的值赋给 a
<code>/=</code>	<code>a/=b</code>	<code>a=a/b</code>	将 a 除以 b 的值赋给 a
<code>%=</code>	<code>a%=b</code>	<code>a=(a%b)</code>	将 a 除以 b 的余数的值赋给 a
<code>&lt;&lt;=</code>	<code>a&lt;&lt;=b</code>	<code>a=(a&lt;&lt;b)</code>	将 a 左移 b 位的值赋给 a
<code>&gt;&gt;=</code>	<code>a&gt;&gt;=b</code>	<code>a=(a&gt;&gt;b)</code>	将 a 右移 b 位的值赋给 a
<code>&amp;=</code>	<code>a&amp;=b</code>	<code>a=(a&amp;b)</code>	将 a 与 b 逐位与的值赋给 a
<code> =</code>	<code>a =b</code>	<code>a=(a b)</code>	将 a 与 b 逐位或的值赋给 a
<code>^=</code>	<code>a^=b</code>	<code>a=(a^b)</code>	将 a 与 b 逐位异或的值赋给 a

赋值运算符的结合性是从右到左的。

【示例 3-8】下列程序代码实现求自然数 1~100 的算术和。在该示例中，使用到了赋值运算符和复合赋值运算符，实现代码及结果如图 3-20 所示。

```
#include<iostream>
using namespace std
int main()
{
    int i,sum;
    sum=0;
    for(i=1;i<=100;i++)           //for循环
    {
        sum+=i;                  //循环相加
    }
    cout<<"运算结果为:" <<sum<<endl; //输出结果
    return 0;
}
```

图 3-20 复合赋值运算符实例

上述程序使用了循环控制结构 `for` 语句，这将在后续章节中详细讲解。此外，代码中使用了复合赋值运算符“`+=`”，并且语句“`s+=i;`”相当于“`s=s+i;`”，即将变量 `s` 本身加上变量 `a` 的值后将结果再赋值给 `s`。

3.4.5 逗号运算符

C++支持逗号运算符的使用。逗号运算符可以使多个表达式写在一行上，从而大大简化了程序，其一般形式及结合性如图 3-21 所示。

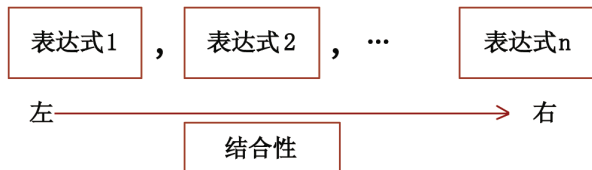


图 3-21 表达式的形式及结合性

表达式的执行顺序如图 3-22 所示。



图 3-22 表达式的执行顺序

【示例 3-10】下面的程序采用了逗号运算符的语句，该语句由 3 个表达式组成，并将其结果赋给整型变量 s，输出 s 的值。实现代码及结果如图 3-23 所示。

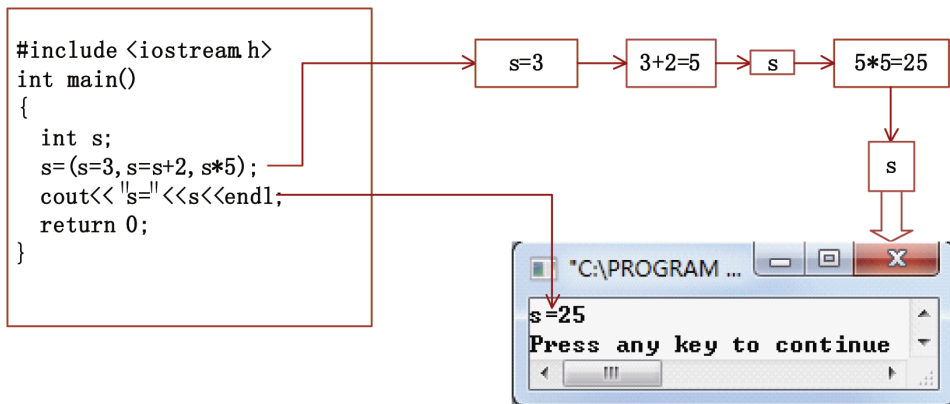


图 3-23 逗号运算符实例

### 3.4.6 sizeof 运算符

由于不同的计算机支持的数据类型长度是不一样的，因此需要一个运算符来测量该机器中的数据类型长度。C++中提供了 sizeof 运算符，其作用和使用格式如图 3-24 所示。

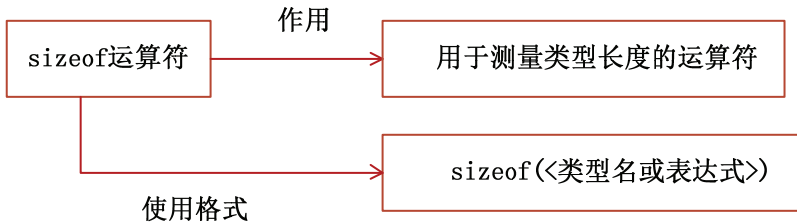


图 3-24 sizeof 运算符的作用和使用格式

该运算符的运算结果是类型名所表示类型的长度或表达式的值所占用的字节数，即这个值所属类型的长度。

【示例 3-11】下面的程序输出在当前测试计算机中几种常用数据类型的长度，其实现代码及结果如图 3-25 所示。



```
#include <iostream>
using namespace std;
int main()
{
    float a;
    int b;
    cout<<"sizeof(shortint)="<<sizeof(short)<<endl;
    cout<<"sizeof(float)="<<sizeof(a)<<endl;
    cout<<"sizeof(int)="<<sizeof(b)<<endl;
    cout<<"sizeof(double)="<<sizeof(double)<<endl;
    cout<<"sizeof(char)="<<sizeof('a')<<endl;
    return 0;
}
```

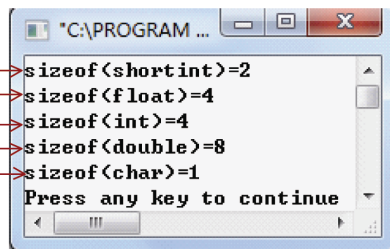


图 3-25 sizeof 运算符实例

### 3.4.7 逻辑运算符

逻辑运算符是作用在逻辑型上的运算，有与（&&）、或（||）、非（!）3 种。逻辑与和逻辑或是双目运算符，逻辑非是单目运算符。

逻辑与运算形式如下：

```
expr1 && expr2;
```

其中，expr1 和 expr2 都是结果为逻辑型值的表达式。当 expr1 和 expr2 都为真时，结果为真；反之则假。

逻辑或运算形式如下：

```
expr1 || expr2;
```

只有当 expr1 和 expr2 都为假时，结果为假；其余为真。

逻辑非运算形式如下：

```
! expr1;
```

非运算的结果与 expr1 相反。



**说明：**①3 种逻辑运算的优先级为“非>与>或”。

②短路表达式：在由&&和||逻辑运算符组成的逻辑表达式中，C++规定，只对能够确定表达式值所需要的最少数目的表达式进行计算。即当计算一个子表达式的值后，可确定整个表达式的值时，后面的表达式便不必再计算了，这种表达式称为短路表达式。

### 3.4.8 运算符的优先级和结合性

前几节简要介绍了运算符的优先级，本节将着重介绍 C++ 的各种运算符之间及同一种类运算符的优先级顺序。

运算符优先级决定了在表达式中各个运算符执行的先后顺序。同一优先级的优先级别相同，运算次序由结合方向决定，使用括号可以改变运算符的顺序。例如，表达式  $1*2/3$ ，\*和/的优先级相同，其结合方向自左向右，则该表达式等价于  $(1*2)/3$ 。

C++ 中，运算符的优先级一般分为 15 级，其优先级中包含的运算符、功能说明和同一优先级的运算符结合性具体如表 3-5 所示。



表 3-5 运算符优先级

优先级	运算符	功能说明	结合性
1	( ) :: [] . , -> . * , -> *	改变优先级 作用域运算符 数组下标 成员选择 成员指针选择	从左至右
2	++, -- & * ! ~ + , - ( ) sizeof new , delete	增 1, 减 1 运算符 取地址 取内容 逻辑求反 按位求反 取正数, 取负数 强制类型 取所占内存字节数 动态存储分配	从右至左
3	*, / , %	乘法, 除法, 取余	从左至右
4	+, -	加法, 减法	从左至右
5	<<, >>	左移位, 右移位	从左至右
6	<, <=, >, >=	小于, 小于等于 大于, 大于等于	从左至右
7	== , !=	相等, 不等于	从左至右
8	&	按位与	从左至右
9	^	按位异或	从左至右
10		按位或	从左至右
11	&&	逻辑与	从左至右
12		逻辑或	从左至右
13	? :	三目运算符	从右至左
14	=, +=, -=, *=, /=, %=, &=, ^=,  =, <<=, >>=	赋值运算符	从右至左
15	,	逗号运算符	从左至右

【示例 3-12】计算下面表达式的值。已知 x、y 是整型，且 x=6, y=10。解答步骤如图 3-26 所示。

$y += x + y \% x << (x -= 2 + y / 2 \& 4)$

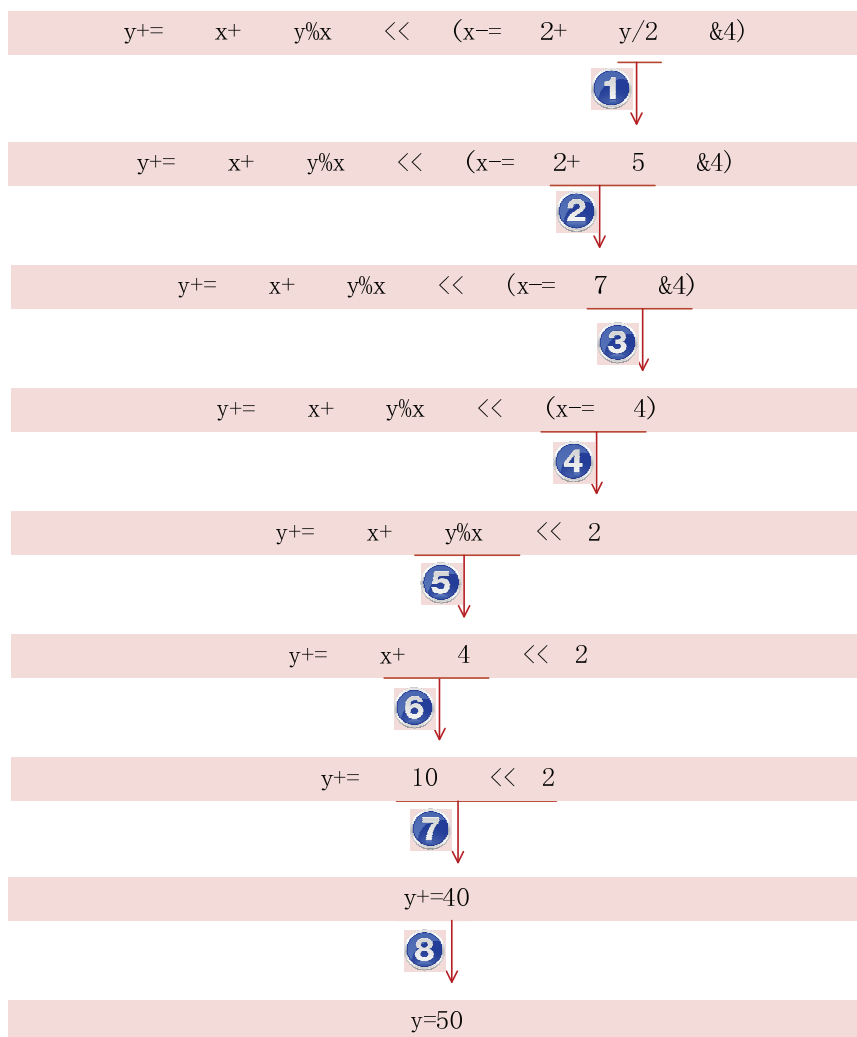


图 3-26 运算符优先级

针对上述表达式编写代码进行验证，代码如下，执行结果如图 3-27 所示。

```
#include<iostream>
using namespace std;
int main()
{
    int x,y;
    x=6;
    y=10;
    y+=x+y%x<<(x-=2+y/2&4);
    cout<<"y= "<<y<<endl;
    return 0;
}
```

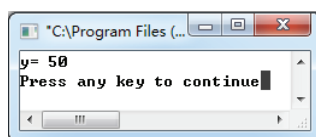


图 3-27 运行结果



## 3.5 语句块

语句块是一个程序片段，它包含多条语句。该片段可以是为了共同完成一个复杂功能而放在一起的，也可以是为了程序的可读性而组织在一起的。本节将介绍语句块的构成及作用域的范围。

### 3.5.1 语句块的构成

在 C++ 程序中，多个连续的语句组成语句块（复合语句）。语句块一般以一对大括号 “{}” 为标志。如图 3-28 所示为一个 main() 函数内的语句块。

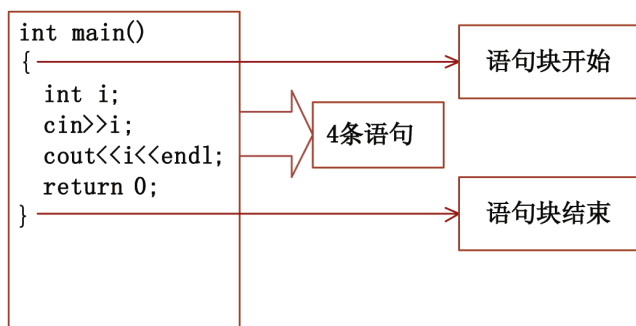


图 3-28 语句块的构成

语句块可以嵌套，如图 3-29 所示。

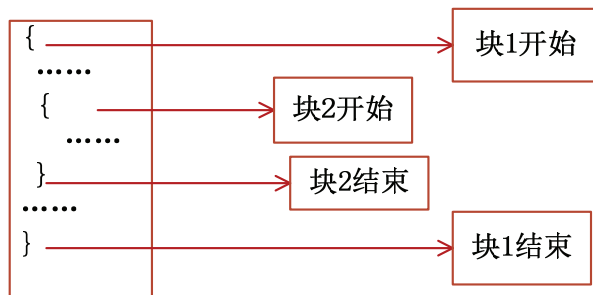


图 3-29 语句块的嵌套



**说明：**当语句块只含有一条语句时，常常可以省去大括号，如果块内包含多条语句，就必须保留大括号，否则意思就完全不一样了。此外，一个好的习惯是先写好一对大括号，再填充语句。

### 3.5.2 作用域——变量的作用范围

在 C++ 语言源文件中，变量的作用域是指变量的有效范围。根据变量是在一对 “{}” 之中还是在程序的所有的 “{}” 之外来确定变量的作用域。根据作用域的不同，变量分为全局变量和局部变量，具体说明事项如图 3-30 所示。

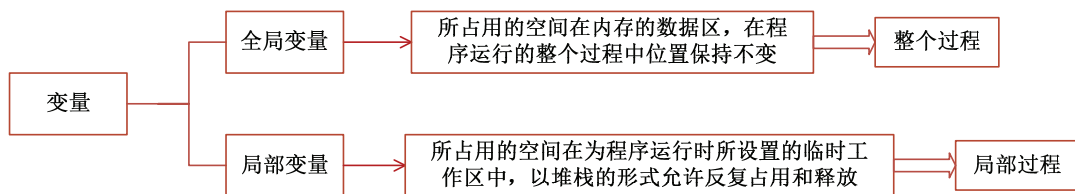


图 3-30 作用域说明

【示例 3-13】该示例通过声明变量语句的位置, 来确定全局变量和局部的作用域。其实现代码及结果如图 3-31 所示。

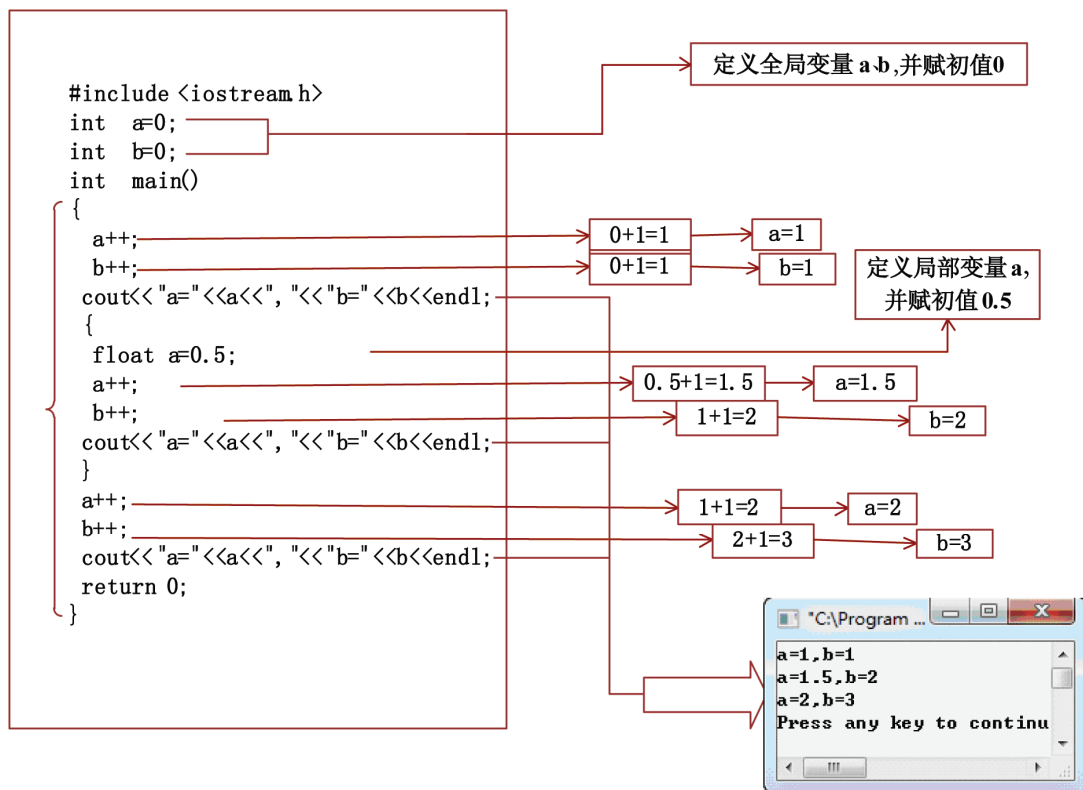


图 3-31 作用域实例

## 3.6 小结

本章主要介绍了 C++程序的基本单位——语句, 以及语句的构成及语句块。重点讲述了 C++中基本运算符的使用及其优先级。并对数据转换和变量的作用域做了简单的介绍。

## 3.7 习题

【题目 3-1】在以下程序代码中, 由于表达式语句的书写中缺少重要成分, 导致程序编译不能通过, 编译器给出的出错信息如图 3-32 所示。请读者找出其中的错误, 并进行改正, 使程序能够正常运行, 输出正确结果。程序的运行效果如图 3-33 所示。程序代码如下:



```
#include<iostream>
using namespace std;
int main()
{
    int a,b,c;
    a=2;
    b=9;
    c=a+b
    cout<<"a+b= "<<c<<endl;
    return 0;
}
```

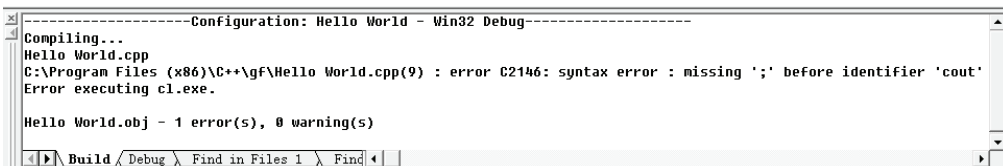


图 3-32 出错信息

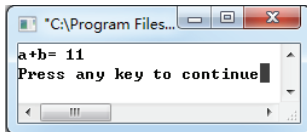


图 3-33 运行效果

【题目分析】本题主要考查表达式语句的正确书写。表达式语句由表达式加分号构成，在书写表达式时很容易丢掉分号，导致编译不能通过。在如图3-32 所示的出错信息中表明确实是语句中缺少了分号。

#### 【关键代码】

```
c=a+b;
```

【题目 3-2】编写代码从键盘接收数据，并将接收的数据输出到屏幕上。要求：输入一个字符'A'、一个整数 26 和一个浮点数 3.14。输出时，一行只要有一个数据，程序的运行效果如图 3-34 所示。

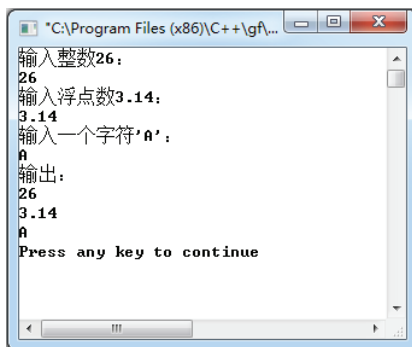


图 3-34 运行效果

【题目分析】本题考查输入/输出语句的使用。题目中要求输出时一行只有一个数据，要达到这样的目的，需在输出一个数据后，就加入一个换行（endl）。

#### 【关键代码】

```
int a;
```





```
double b;
char c;
cout<<"输入整数 26: "<<endl;
cin>>a;
cout<<"输入浮点数 3.14: "<<endl;
cin>>b;
cout<<"输入一个字符'A': "<<endl;
cin>>c;
cout<<"输出: "<<endl<<a<<endl<<b<<endl<<c<<endl;
```

【题目 3-3】下面的程序代码因为赋值运算符的使用错误，使程序不能正常运行。出错信息如图 3-35 所示，请读者找出问题并改正。程序代码如下：

```
#include<iostream>
using namespace std;
int main()
{
    const int a=26;
    int b,c;
    b=5
    a=a+b;
    c=a*a;
    cout<<" (a+b) 的平方为: "<<c<<endl;
    return 0;
}
```

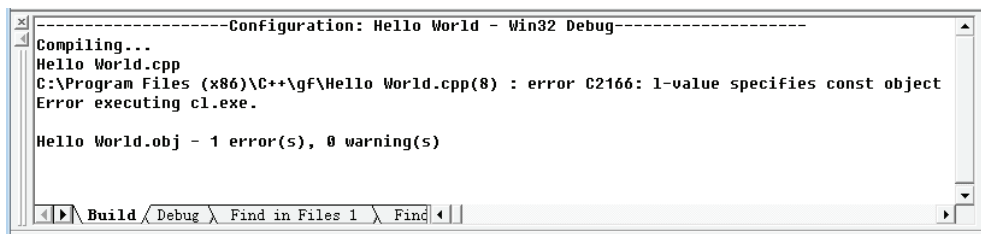


图 3-35 出错信息

【题目分析】本题主要考查赋值运算符的使用。赋值运算符左边的操作数必须是可以修改的变量，不能是常量。

【关键代码】

```
b=a+b;
c=b*b;
```

【题目 3-4】输入一个字符和一个正整数，并求该字符的 ASCII 码和这个整数的和。要求：结果要在显示器上显示，效果如图 3-36 所示。

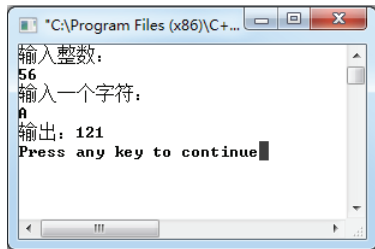


图 3-36 运行效果

【题目分析】本题考查隐式转换的知识。字符型数据是以 ASCII 码的形式存储的。字符型



数据和整型数据混合运算时，字符型数据会隐式进行转换。

**【关键代码】**

```
int a;
char c;
cout<<"输入整数: "<<endl;
cin>>a;
cout<<"输入一个字符: "<<endl;
cin>>c;
a=c+a;
cout<<"输出: "<<a<<endl;
```

**【题目 3-5】**从键盘输入两个数  $x$  和  $y$ ，用复合赋值运算符计算  $x+y$ 、 $x*y$ 、 $x\%y$  的值。

**【题目分析】**主要考查复合赋值运算符的使用方法。

**【关键代码】**

```
int x,y;
cin>>x;
cin>>y;
cout<<"(x+=y)<<endl;
cout<<"(x*=y)<<endl;
cout<<"(x%=y)<<endl;
```

**【题目 3-6】**设华氏温度为  $f$ ，摄氏温度为  $c$ ，则摄氏温度到华氏温度的转换公式为：  
 $f=c*9/5+32$ 。

编程实现上述表达式。要求：从键盘输入摄氏温度，然后将摄氏温度转换为华氏温度输出到显示器上，运行效果如图 3-37 所示。

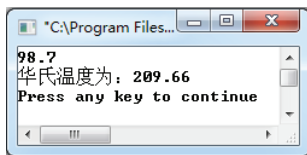


图 3-37 运行效果

**【题目分析】**主要考查表达式的用法。

**【关键代码】**

```
double c,f;
cin>>c;
f=c*9/5+32;
cout<<"华氏温度为: "<<f<<endl;
```

**【题目 3-7】**利用算术运算符计算  $(100+303*630-90/4)/24$  的余数。

**【题目分析】**主要考查算术运算符的相关知识，重点掌握算术运算符的优先级及使用技术。

**【关键代码】**

```
float a=0;
a=(100+303+630-290/4)/24;
cout<<"a="<<a<<endl;
```

**【题目 3-8】**计算下面表达式的值。已知  $k$ 、 $i$  是整型，且  $k=1$ ， $i=-22$ 。要求：计算表达式的值时，需要写详细的解答步骤。

$k=(k+=-++i/3*2>>1)*2+10$

**【题目分析】**本题主要考查运算符的优先级和结合性知识。



【关键步骤】表达式的求解步骤如图 3-38 所示。

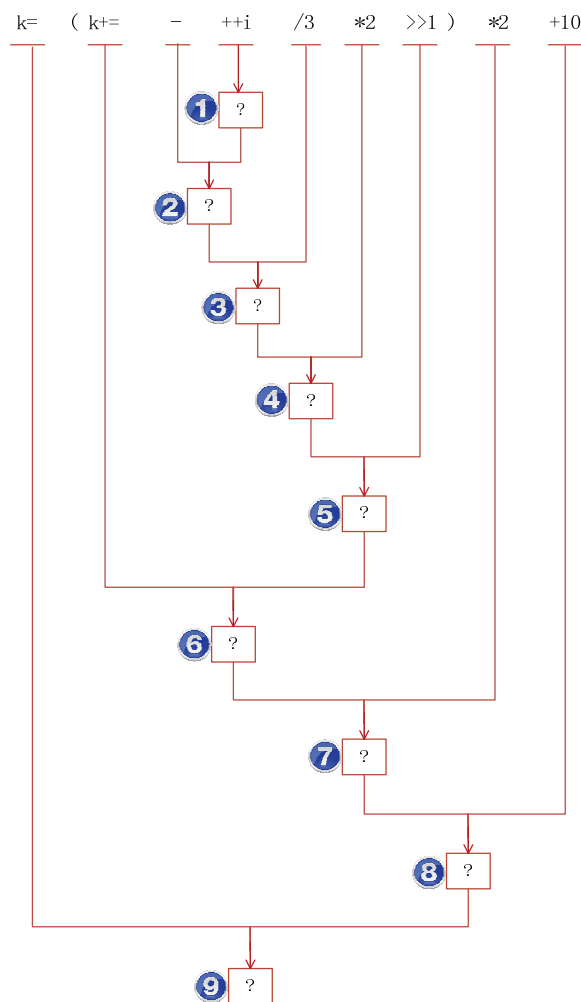


图 3-38 运算符的优先级

【题目 3-9】下面的代码输出结果如图 3-39 所示。根据输出结果，考虑为什么会输出这样的结果。

```
int main()
{
    int i=10;
    int k=1;
    //语句 1
    k+=i/=2;
    cout<<"k= "<<k<<endl;
    //语句 2
    ++--k;
    cout<<"k= "<<k<<endl;
    //语句 3
    cout<<(k<=i>=0)<<endl;
    return 0;
}
```

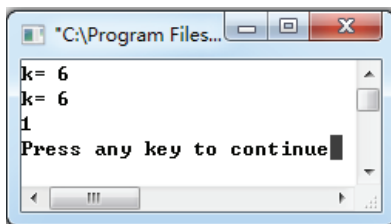


图 3-39 运行结果

【题目分析】本题主要考查运算符的结合性对计算结果的影响。

【关键步骤】

语句 1 中包含+=、/=运算符，两者级别相同，但结合性为从右到左。

语句 2 中++和--结合性是从右到左，故先算++再算--。

语句 3 中<=和>=的结合性是从左到右。

【题目 3-10】一个圆球的半径  $r$  是 96.126 厘米，求这个圆球的表面积  $s$  和体积  $v$ ，下面给出了球体的体积公式和表面积公式。要求：编程计算表面积和体积，求得的结果中不要含有小数部分，只需要整数部分。程序运行效果如图 3-40 所示。

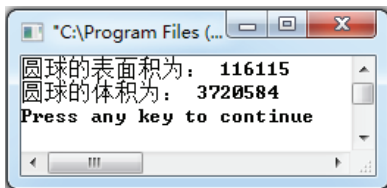


图 3-40 运行效果

球体的体积公式： $v=4*PI*r*r*r/3$ ;

球体的表面积公式： $s=4*PI*r*r$ 。

其中，PI 是圆周率常数， $PI=3.1415926$ 。

【题目分析】本题中给出的数据都是浮点型的，而要求结果为整型，这就需要用到类型转换。

【关键代码】

```
double v,s, r=96.126;
const double PI=3.1415926;
v=4.0*PI*r*r*r/3.0;
s=4*PI*r*r;
cout<<"圆球的表面积为: "<< static_cast<int>(s)<<endl;
cout<<"圆球的体积为: "<< static_cast<int>(v)<<endl;
```

【题目 3-11】在不使用临时变量的情况下，用按位异或运算交换两个整型变量的值。要求：定义两个整型变量，整型变量中的值由用户从键盘输入，然后将交换后的值输出到显示器。程序运行效果如图 3-41 所示。

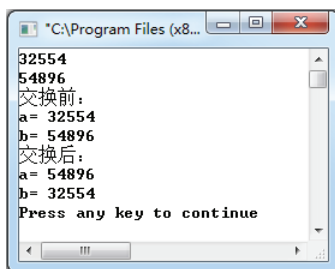


图 3-41 运行效果

【题目分析】本题考查位运算符的使用。这里定义两个整型变量  $a$ 、 $b$  并初始化。首先， $a$  和  $b$  进行异或运算，结果赋给  $a$ 。如此  $a$  中就相当于两个值合并信息。再让  $a$ 、 $b$  进行异或运算，结果赋给  $b$ ，此时就从  $a$  中分离出了原来  $a$  中的值。这时， $b$  中存放的就是  $a$  原来的值。同理，再对两者进行运算，分离出  $b$ ，存储到  $a$  中。

算法：将  $a$ 、 $b$  进行三次异或运算。

```
a=a^b; b=a^b; a=a^b;
```

#### 【关键代码】

```
int a,b;  
cin>>a>>b;  
cout<<"交换前: "<<endl;  
cout<<"a= "<<a<<endl;  
cout<<"b= "<<b<<endl;  
a=a^b;  
b=a^b;  
a=a^b;  
cout<<"交换后: "<<endl;  
cout<<"a= "<<a<<endl;  
cout<<"b= "<<b<<endl;
```

# 第4章 程序控制结构

程序控制结构即结构化程序设计用到的结构，或者称为流程控制结构。C++具有结构化程序设计的4种结构：顺序结构、选择结构、循环结构、转向结构。本章将着重介绍这几种结构在C++中的流程控制语句及其实现。

## 4.1 语句块的执行方式——顺序结构

顺序结构是指按照所有语句出现的顺序先后执行，先出现的先执行，后出现的后执行。顺序结构的执行流程如图4-1所示。



图 4-1 顺序结构

前面所讲的表达式语句、输入语句和输出语句都是顺序结构的语句。

## 4.2 条件的表达

“人生不如意，十有八九”，程序也不都完全是顺序执行的。有时需要根据实际情况选择执行，即后面所要讲的选择结构、循环结构和转向结构。结构的选择需要条件的表达，本节将介绍两种条件的表达：关系运算符和逻辑运算符。

### 4.2.1 单一条件的表达——关系运算符

关系运算符是双目运算符，其作用是将两个运算分量进行大小比较，其运算结果类型为布尔数据类型。若关系成立，则值为 `true`，否则为 `false`。C++中支持的关系运算符主要有6种，其运算符、名称、功能，以及各运算符的相关示例如表4-1所示。



表 4-1 关系运算符

运 算 符	运算符名称	功 能	示 例	结 果
<	小于	若 a<b, 结果为 true, 否则为 false	2<3	true
<=	小于等于	若 a<=b, 结果为 true, 否则为 false	7<=3	false
>	大于	若 a>b, 结果为 true, 否则为 false	7>3	true
>=	大于等于	若 a>=b, 结果为 true, 否则为 false	3>=3	true
==	等于	若 a==b, 结果为 true, 否则为 false	7==3	false
!=	不等于	若 a!=b, 结果为 true, 否则为 false	7!=3	true

关系运算符的结合性是从左到右的, 优先级如图 4-2 所示。



图 4-2 关系运算符的优先级

【示例 4-1】使用关系运算符进行两个操作数的比较, 将结果赋值给布尔型变量, 并输出该变量的值。其实现代码及结果如图 4-3 所示。

```

#include <iostream>
int main()
{
    int a=3,b=5;           //定义变量并初始化
    bool k;
    k=(a>b);               //关系运算
    cout<<"k="<<k<<endl;  //输出结果
    k=(a<b);               //关系运算
    cout<<"k="<<k<<endl;  //输出结果
    return 0;
}

```

图 4-3 关系运算符示例

关系表达式“a>b”, 3 比 5 要小, 表达式的值为假, 所以首先输出 k 的值为 0。同理得出“a<b”的值。

### 4.2.2 多条件的表达——逻辑运算符

多个单一条件组合在一起就构成了多条件, 这时组合单一条件时需要的是逻辑运算符。C++提供了 3 种逻辑运算符, 单目运算符逻辑非 (!)、双目运算符逻辑与 (&&) 和逻辑或 (||)。运算结果类型为布尔型数据类型, 其值为 true 或 false。这 3 种运算符符号、名称、功能及其相关示例如表 4-2 所示。

表 4-2 逻辑运算符

运 算 符	运算符名称	功 能	示 例	结 果
!	逻辑非	当运算分量为 false 时, 结果为 true 当运算分量为 true 时, 结果为 false	!0 !1	true false
&&	逻辑与	当两个运算分量都为 true 时, 结果才为 true	0&&0 0&&1 1&&1	false false true
	逻辑或	当两个运算分量有一个为 true 时, 结果就为 true	0  0 0  1 1  1	false true true



逻辑运算符的结合性是从左到右的，优先级如图 4-4 所示。



图 4-4 逻辑运算符的优先级

【示例 4-2】下面的程序给出了几个逻辑运算表达式，将其值赋给布尔型变量，并输出该变量的值，其实现代码及结果如图 4-5 所示。

```
#include <iostream.h>
int main()
{
    int a=3,b=5;           //定义变量并初始化
    bool k;
    k=((a<b) || (a>b));    //逻辑或运算
    cout<<"k="<<k<<endl;  //输出结果
    k=((a<b)&&(a>b));      //逻辑与运算
    cout<<"k="<<k<<endl;  //输出结果
    return 0;
}
```

图 4-5 逻辑运算符示例

在【示例 4-1】中，“a<b”的值为真，“a>b”的值为假，对这两个关系表达式进行逻辑或运算，结果为真，进行逻辑与运算结果为假。



## 4.3 选择结构

选择结构是用来判断所给定的语句是否满足条件，根据判断结果，选择执行不同的分支语句。常用的有：条件运算符、if 语句、if...else 语句、多重 if...else 语句和 switch 语句。

### 4.3.1 最简单的选择——条件运算符

C++中，条件运算符是一个比较特殊的运算符，其是双目运算符，说明语句的一般形式及结合性如图 4-6 所示。

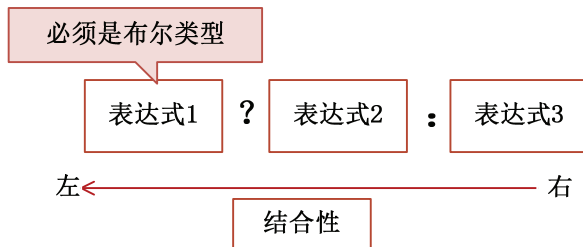


图 4-6 条件运算符的形式及结合性





表达式的执行顺序如图 4-7 所示。

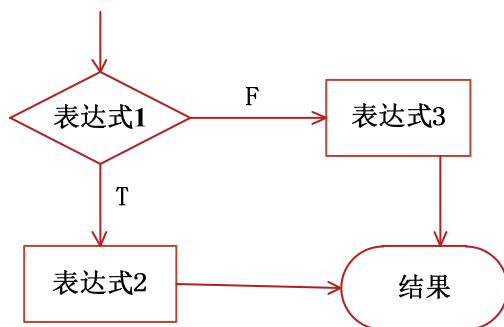


图 4-7 表达式的执行顺序

【示例 4-3】根据用户输入的两个数值，比较其值，并输出其中大的数值，实现代码及结果如图 4-8 所示。

```

#include <iostream h>
int main()
{
    int a, b, x;           //定义变量
    cout<<"请输入两个数值:" <<endl;
    cin>>a>>b;             //接收用户输入
    x=a>b?a:b;             //条件运算符
    cout<<"大的数值是:" <<x<<endl //输出结果
    return 0;
}
  
```

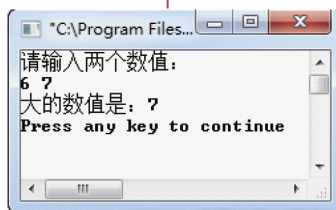
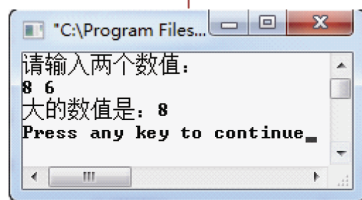


图 4-8 条件运算符示例

“8>6”为真，所以条件表达式的值为 8，从而输出结果为 8；“6>7”为假，所以条件表达式的值为 7，从而输出为 7。

### 4.3.2 单分支条件语句——if 语句

if 语句为单分支条件语句，其说明语句的一般形式如图 4-9 所示。

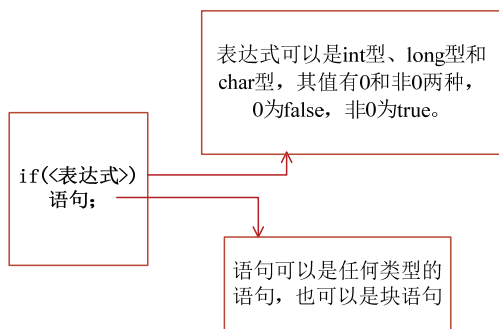


图 4-9 if 语句的一般形式

该语句的作用是：如果表达式的值为 true，则执行 if 后面的语句；否则跳过执行后面的语句。其执行顺序如图 4-10 所示。

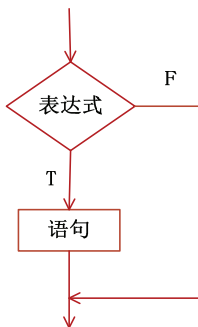


图 4-10 if 语句执行流程

【示例 4-4】已知两个变量 x 和 y，比较它们的大小，当 x 小于 y 时，交换这两个变量的值，使 x 的值大于 y。实现程序代码及结果如图 4-11 所示。

```
#include <iostream.h>
int main()
{
    int x,y,temp;
    cout<<"Please input 2 numbers:";
    cin>>x>>y;
    if (x<y)
    {
        temp=x;
        x=y;
        y=temp;
    }
    cout<<"x="<<x<<endl;
    cout<<"y="<<y<<endl;
    return 0;
}
```

定义变量

接收用户输入

判断x、y的大小

x小于y则交换

图 4-11 if 语句示例



**注意：**if 语句一般用于有多种可能情况，但只有一种情况需要进行操作，即其他情况不需要进行程序控制的情况下。



### 4.3.3 双分支条件语句——if...else 语句

if...else 语句为双分支条件语句，其说明语句的一般形式为：

```
if (<表达式>)  
    <语句 1>;  
else  
    <语句 2>;
```

该语句的作用是：如果表达式的值为 true，则执行“语句 1”；否则执行“语句 2”。相对于如上的 if 语句，if...else 语句增加了对于表达式的值为 false 时的处理语句，其执行流程如图 4-12 所示。

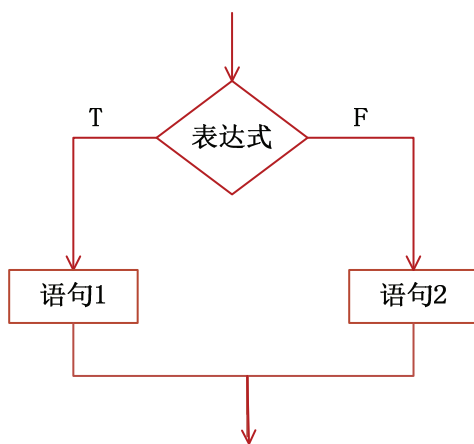


图 4-12 if...else 语句的执行流程

【示例 4-5】该示例根据输入的学生的百分制成绩，判断该学生是否及格，其实现程序代码及结果如图 4-13 所示。

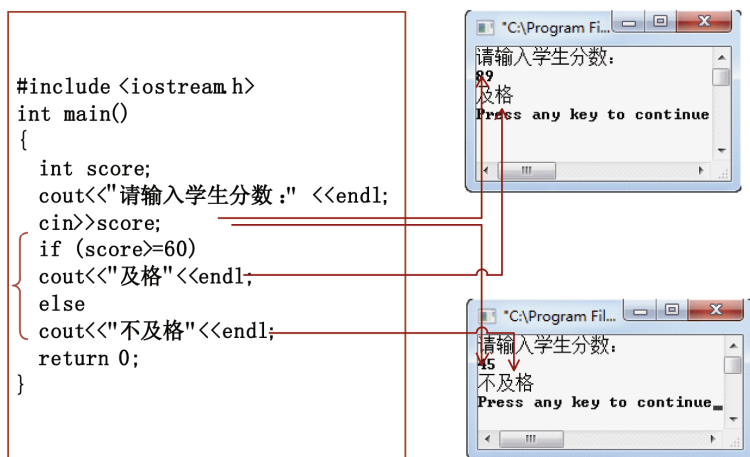


图 4-13 if...else 语句示例



**提示：**if...else 语句一般用于有两种情况，其一，根据其是否满足条件而分别执行不同的程序段，其二在进行不同操作的情况下。



### 4.3.4 多重 if...else 语句

多重 if...else 语句为多分支条件语句或 if...else if...else 语句，其说明语句的一般形式及执行流程如图 4-14 所示。

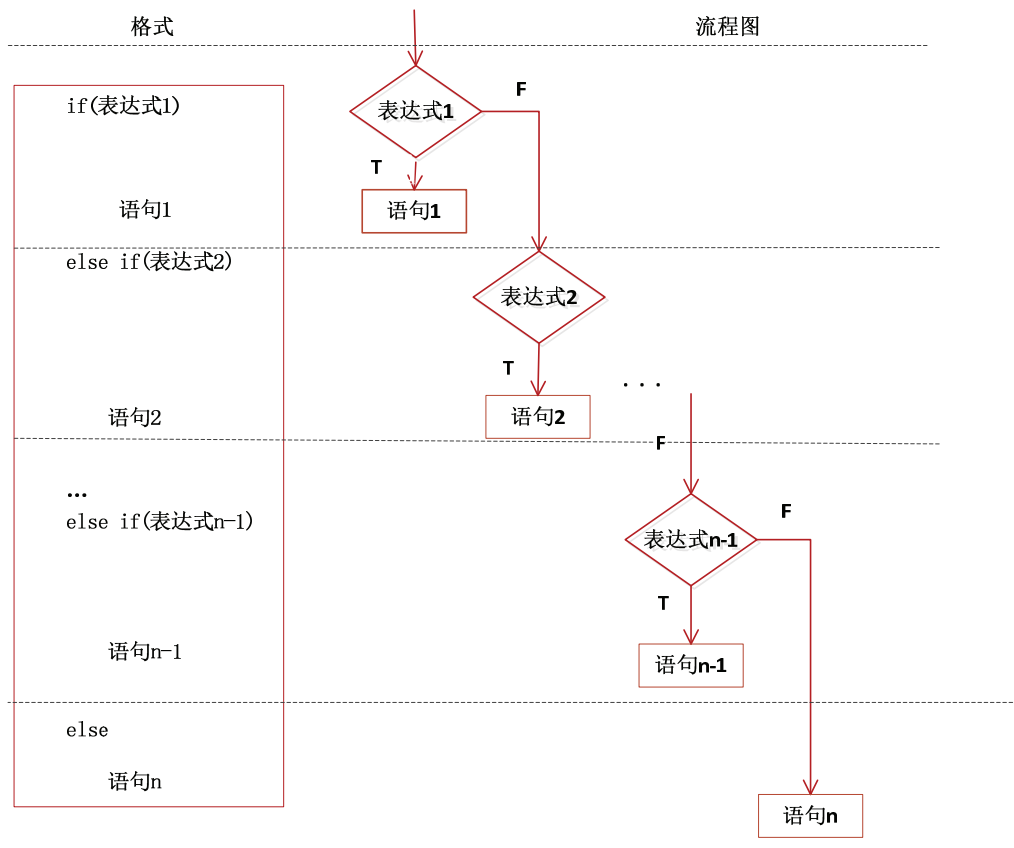


图 4-14 多重 if...else 语句的一般形式及执行流程图

该语句的作用是：首先测试表达式 1，如果它为 false，接着就测试表达式 2，依次类推，直到找到一个为 true 的条件就执行相应的语句，执行后即跳出该语句。如果所有条件都不是 true，则执行 else 语句。



**注意：**该语句为 if 语句的嵌套，在嵌套时，遵守就近原则，即 C++ 语言规定每个 else 只与其前面最近的未配对的 if 配对，也可以用 { } 确定层次关系。

【示例 4-6】该示例输入学生的百分制成绩，判断该学生的等级。其等级评定条件如图 4-15 所示。

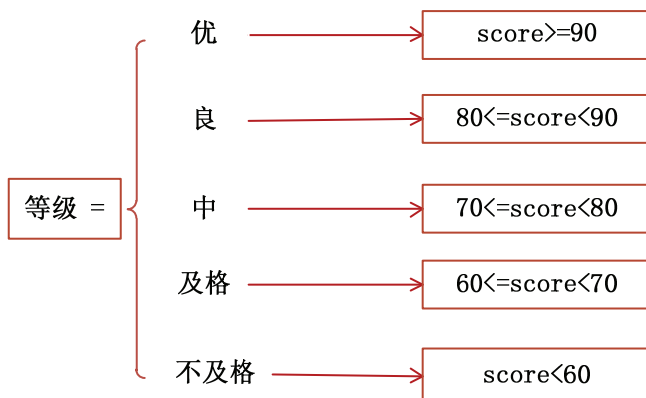


图 4-15 score 等级评定条件

可以看出该示例的等级有 5 种情况，因此可以使用多重分支语句 if...else if...else 来实现。其实现程序代码及结果如图 4-16 所示。

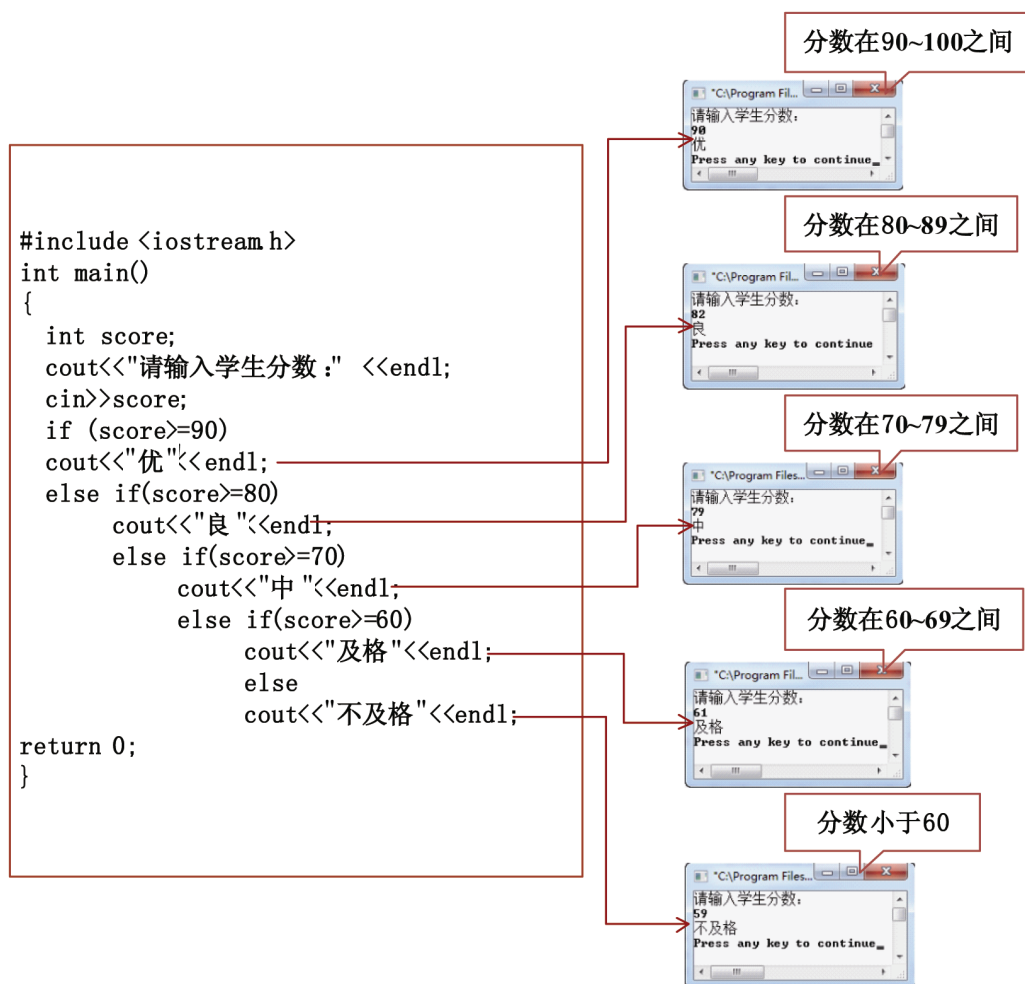


图 4-16 多重 if...else 语句示例



### 4.3.5 情况语句——switch 语句

switch 语句也称情况语句，其也是一种多分支语句，其说明语句的一般形式及执行流程如图 4-17 所示。

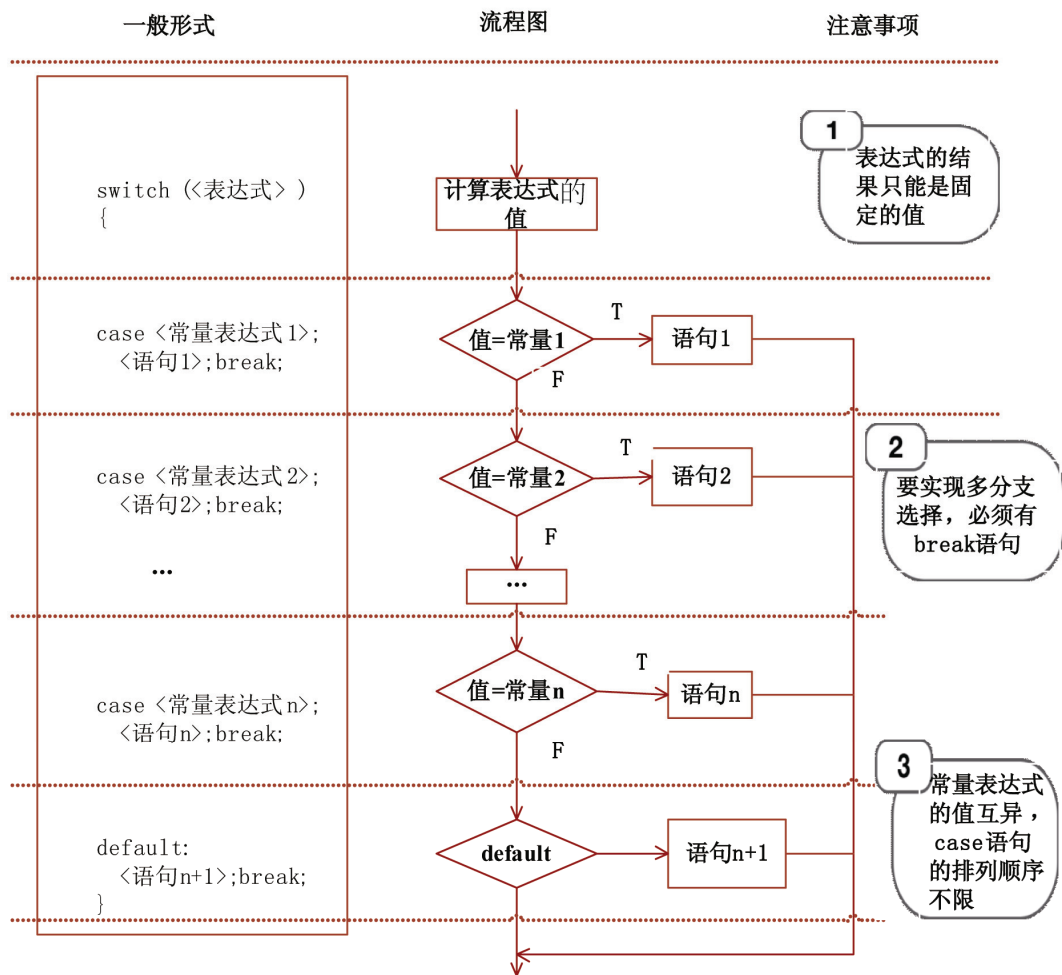


图 4-17 switch 语句的一般形式及执行流程图

该语句的作用是：首先计算 switch 表达式后的值，然后将其结果与 case 后面的各常量表达式进行比较，若匹配，则执行该分支后的语句，执行完后，遇到 break 语句，则退出 switch 语句；若其结果与 case 后面的各常量表达式都不匹配，则执行 default 后面的语句。

【示例 4-7】该示例输入 0~6 之间的整数，转换成星期输出，其实现代码及结果如图 4-18 所示。



```
#include <iostream.h>
```

```
int main ()
```

```
{
```

```
    int day;
```

```
    cout<<"请输入0~6的数值:" <<endl;
```

```
    cin>>day;
```

```
    switch(day)
```

```
    {
```

```
        case 0:
```

```
        cout<<"星期日"<<endl; break;
```

```
        case 1:
```

```
        cout<<"星期一"<<endl; break;
```

```
        case 2:
```

```
        cout<<"星期二"<<endl; break;
```

```
        case 3:
```

```
        cout<<"星期三"<<endl; break;
```

```
        case 4:
```

```
        cout<<"星期四"<<endl; break;
```

```
        case 5:
```

```
        cout<<"星期五"<<endl; break;
```

```
        case 6:
```

```
        cout<<"星期六"<<endl; break;
```

```
        default:
```

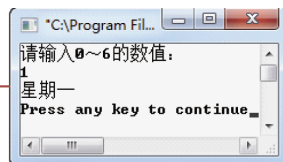
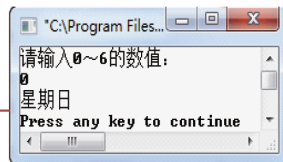
```
        cout<<"输入非法"<<endl;
```

```
    }
```

```
    return 0;
```

```
}
```

用户输入



.....

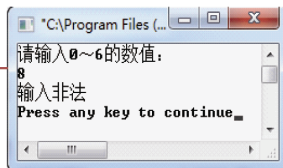
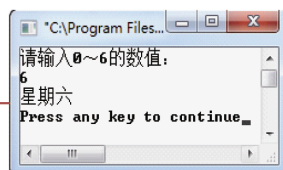


图 4-18 switch 语句示例



**警告：**在使用 switch...case 语句时，一定要注意使用 break 语句，否则将因无法跳出分支不能继续执行下去，从而导致输出错误。

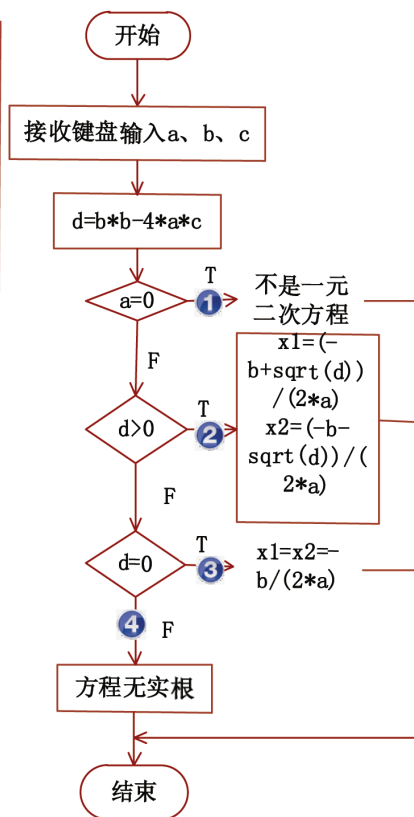
### 4.3.6 应用实例

4.3.1~4.3.5 节详细讲解了选择结构的实现语句，本小节将通过一个综合示例回顾一下选择结构的实现。

【示例 4-8】本示例求方程  $ax^2+bx+c=0$  的根。该程序的实现需要接收用户输入  $a$ 、 $b$ 、 $c$ ，并根据输入判断是否有实根并求出解，实现代码、执行流程及结果如图 4-19 所示。



```
#include <iostream.h>
#include <math.h>
int main()
{
    float a, b, c, d;
    cout<<"a="; cin>>a;
    cout<<"b="; cin>>b;
    cout<<"c="; cin>>c;
    d=b*b-4*a*c;
    if (a==0)
        cout<<"不是一元二次方程"<<endl;
    else if (d>0)
    {
        cout<<"方程有两个实根!"<<endl;
        cout<<"x1="<<(-b+sqrt(d))/(2*a)<<endl;
        cout<<"x2="<<(-b-sqrt(d))/(2*a)<<endl;
    }
    else if(d==0)
    {
        cout<<"方程有一个实根!"<<endl;
        cout<<"x1=x2="<<-b/(2*a)<<endl;
    }
    else
        cout<<"方程无实根!"<<endl;
    return 0;
}
```

图 4-19 方程  $ax^2+bx+c=0$ 

关于浮点数的比较:

```
int main()
{
    double x=1.0;
    double y=3.0/7.0+2.0/7.0+2.0/7.0;
    cout<<y<<endl;
}
```

输出: 0.999999

浮点数在计算机中的存储是近似的, 比较时只要两个浮点数近似相等就认为它们相等。



## 4.4 循环结构

循环结构用来在指定的条件下多次重复执行同一组语句。在 C++ 中, 常用的循环语句形式主要有 3 种: for 语句、while 语句、do...while 语句。





### 4.4.1 for 语句

for 语句是 C++ 中最常见的、功能最强的循环语句。它既可用于循环次数确定的情况，也可用于循环次数不确定而只给出循环结束条件的情况，其说明语句的一般形式及执行流程如图 4-20 所示。

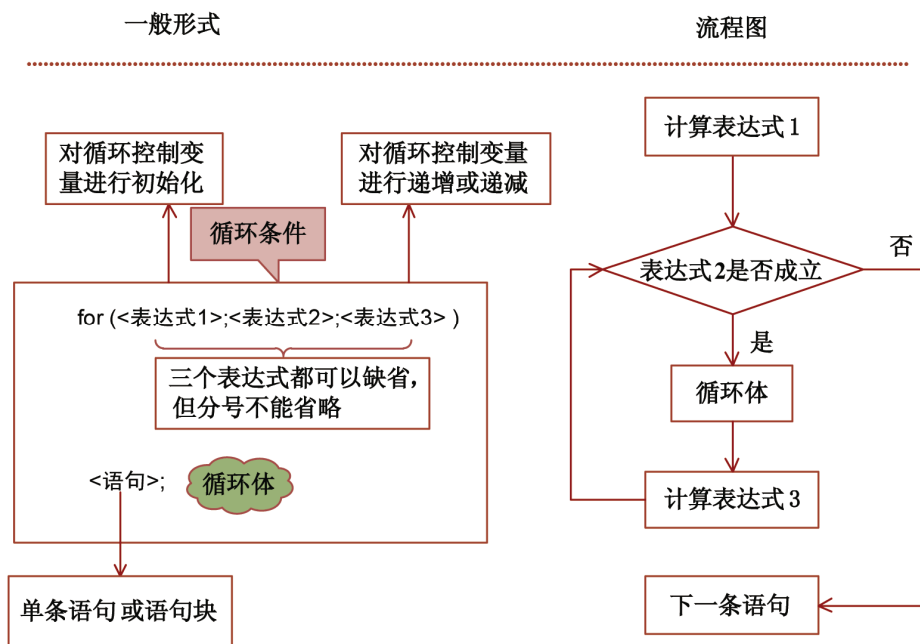


图 4-20 for 语句的一般形式及执行流程

【示例 4-9】该示例求自然数 100 内的所有偶数之和，即计算  $2+4+6+\cdots+100$  的算术和。其实现代码与结果如图 4-21 所示。

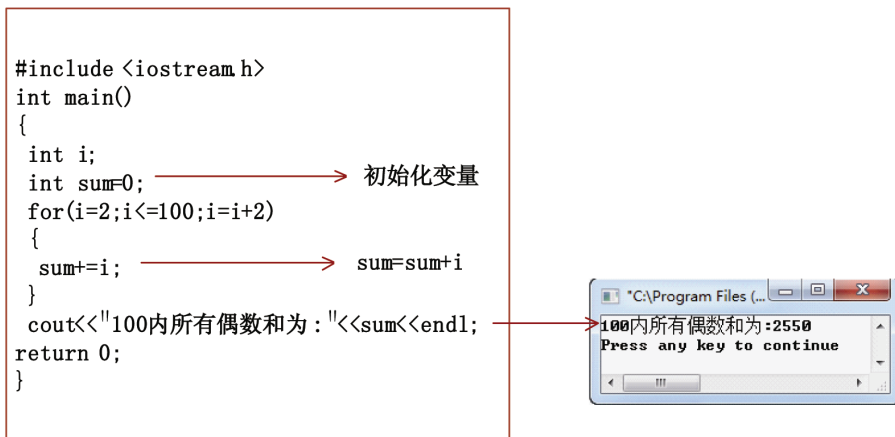


图 4-21 100 内偶数之和



**警告：**for 语句中包含 3 个语句，这是必不可少的，即使其中某一个语句为空，“;”是不能省略的，否则编译系统将给出错误提示。



## 4.4.2 while 语句

while 语句是最简单的循环语句, 它实际上是 for 语句的表达式 1 和表达式 3 为空的特殊情形, 其说明语句的一般形式及执行流程如图 4-22 所示。

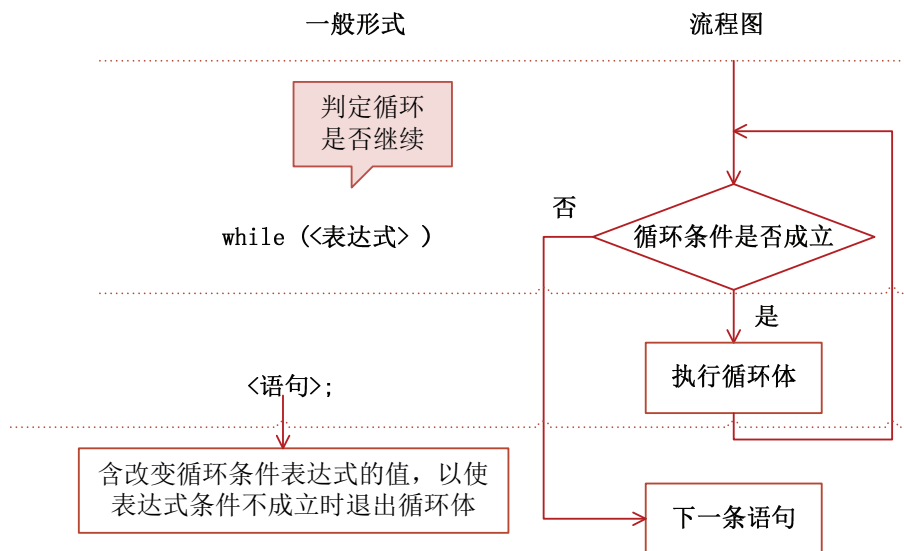


图 4-22 while 语句的一般形式及执行流程

使用 while 语句也同样可以实现类似 for 语句的循环。

【示例 4-10】将【示例 4-9】中求 100 以内所有偶数和通过 while 语句来实现。其实现代码及结果如图 4-23 所示。

```
#include <iostream.h>  
int main()  
{  
    int i;  
    int sum=0; —————> 初始化变量  
    i=2;  
    while(i<=100)  
    {  
        sum+=i; —————> sum=sum+i  
        i=i+2;  
    }  
    cout<<"100内所有偶数和为:"<<sum<<endl;  
    return 0;  
}
```

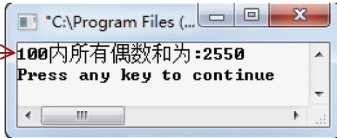


图 4-23 while 语句示例



**注意:** 在 while 语句的循环体中, 一定要包含如 i++之类的循环变量递增或递减的语句, 以使其经过若干次循环后不满足循环条件, 从而跳出循环。否则, 程序将陷入死循环。



while 循环的几种形式如下。

(1) 计数控制。在循环中设置一个变量作为计数器，执行一次循环则计数器加一，多用于循环次数确定的情况。一般形式如下：

```
counter=0;
while (counter<n)
{
    ...
    counter++;
    ...
}
```

(2) 步哨控制。循环结束条件是一个特殊值，这个特殊值称为步哨，多用于循环次数无法确定的情况。其一般格式如下：

```
cin>>i;
while (i!=sentinel)    //sentinel 为一个确定的值
{
    cin>>i;
}
```

(3) 标志控制。先定义一个布尔型变量，用这个布尔型变量控制循环。其一般格式如下：

```
bool f=false;
while (!f)
{
    if (expression)
        f=true;
}
```

(4) EOF 控制。其一般格式如下：

```
cin>>i;
while (cin)
{
    cin>>i;
}
```

#### 4.4.3 do...while 语句

do...while 语句是 while 语句的一种变化形式，其说明语句的一般形式及执行流程如图 4-24 所示。

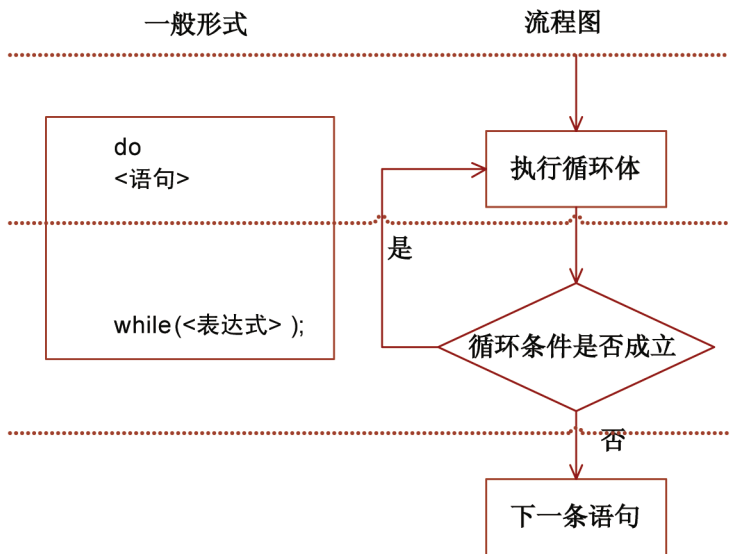


图 4-24 do...while 语句的一般形式及执行流程

do...while 语句与 while 语句的主要区别是：do...while 语句的循环体至少被执行一次，而 while 语句先判断条件，有可能一次也不执行。

【示例 4-11】将【示例 4-9】中求 100 以内所有偶数和通过 do...while 语句来实现。其实现代码及结果如图 4-25 所示。

```
#include <iostream.h>
int main()
{
    int i;
    int sum=0;
    i=2;
    do
    {
        sum+=i;
        i=i+2;
    }
    while(i<=100);
    cout<<"100内所有偶数和为："<<sum<<endl;
    return 0;
}
```

初始化变量

sum=sum+i

图 4-25 do...while 语句示例

#### 4.4.4 多重循环

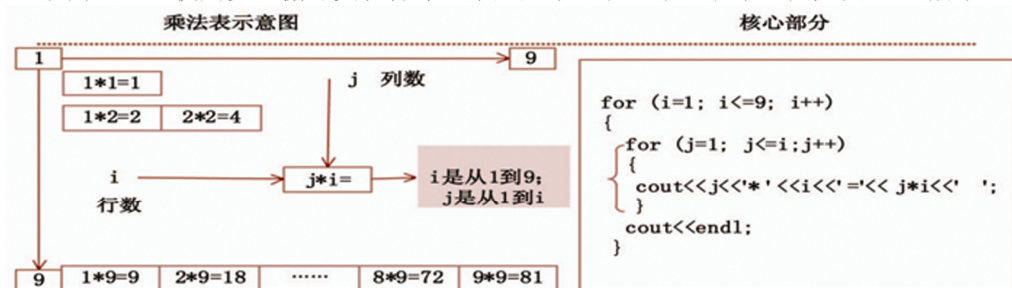
在实际应用中，还有一种循环方式使用很广泛，这就是多重循环，也称为循环嵌套，是指循环语句的循环体内又包含另一个循环语句。



**注意：**在多重循环中，循环嵌套的执行顺序是先执行最里层的循环语句，依次往外执行，最后执行最外层的循环。



【示例 4-12】使用多重循环实现打印一个九九乘法表。乘法表示意图如图 4-26 所示。



4-26 乘法表示意图

根据上述分析，乘法表的完整实现代码及结果如图 4-27 所示。

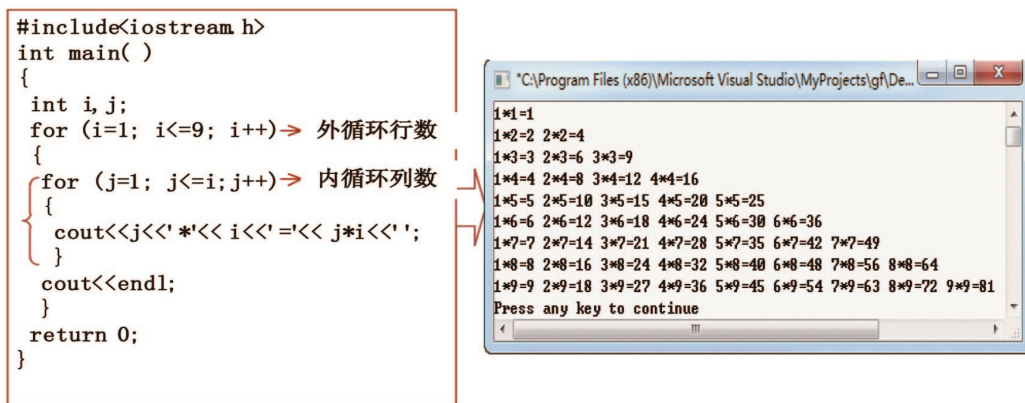


图 4-27 九九乘法表

总的来说，for 语句、while 语句和 do...while 语句都能够实现循环结构。但是在 C++中，最常用的还是 for 语句，这是因为它集成了 3 个语句在其中，写法精练。当然，读者可以根据具体情况选择使用不同的语句。

while 循环和 for 循环先判断循环条件，再执行循环体，do\_while 循环是先依次执行循环体，再判断循环条件。故 while 循环和 for 循环又称先判断循环，do\_while 循环又称后判断循环。

#### 4.4.5 应用举例

【示例 4-13】利用 for 循环，获取 100 以内 2 的幂。程序代码如下，程序运行结果如图 4-28 所示。

```
#include<iostream>
#include<cmath>
using namespace std;
int main()
{
    for(int power=1;power<=100;power*=2) //利用 for 循环求 2 的幂，限定 100 以内
    {
        cout<<power<<endl;                //输出 2 的幂
    }

    return 0;
}
```

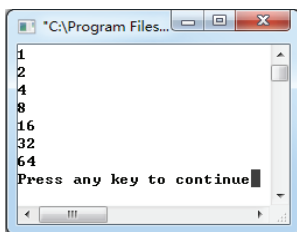


图 4-28 100 以内 2 的幂

在示例代码的 for 循环语句中，初始化语句将变量 power 初始化为 1，也就是 2 的 0 次幂。条件表达式 power<=100 限定这个幂是 100 以内的值。而动作语句 power\*=2 则依次求 2 的幂，即后一个幂是前一个幂的 2 倍。



## 4.5 意外情况的表达——转向语句

在程序控制过程中，有时需要借助特殊的手段——转向语句来实现。转向语句是 C++ 中用来实现无条件转移的语句。常用的转向语句有 break、continue、goto。

### 4.5.1 跳出语句——break 语句

break 语句又称跳出语句，用来结束循环结构，然后执行循环体后面的语句，其说明语句的一般形式如下：

```
break;
```

break 语句通常与一个 if 语句配合使用。

【示例 4-14】程序中使用 break 语句终止整个循环。其实现代码及结果如图 4-29 所示。

```
#include <iostream.h>
int main()
{
    for(int i=0;i<5;i++)
    {
        if(i==2)
            break;
        cout<<i<<endl;
    }
    return 0;
}
```

当i的值是2  
时运行break  
语句，结束  
整个循环

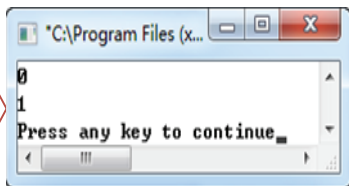


图 4-29 break 语句

break 语句也可以作为 switch 语句的出口，用于退出 case 语句，这在 switch 语句中已用过。



**注意：**break 语句只能用在 switch 语句和循环语句中，否则会引起编译错误。

### 4.5.2 继续语句——continue 语句

continue 语句又称继续语句，可用来跳出本次循环而进入下一次循环，其说明语句的一般形式如下：



continue;

continue 语句与 break 语句的主要区别是：continue 语句是根据条件判断只结束本次循环，不结束整个循环结构；而 break 语句是结束整个循环结构，然后执行循环体后面的语句。

【示例 4-15】程序中使用 continue 语句结束本次循环。其实现代码及结果如图 4-30 所示。

```
#include <iostream.h>
int main()
{
    for(int i=0;i<5;i++)
    {
        if(i==2)
            continue;
        cout<<i<<endl;
    }
    return 0;
}
```

当i的值是2时，  
运行continue  
语句，结束本  
次循环，进入  
下一次

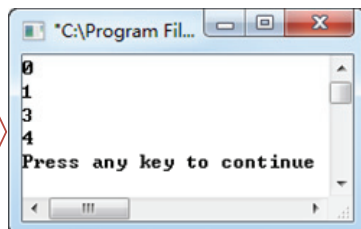


图 4-30 continue 语句



**注意：**continue 语句只能用在循环语句中。

### 4.5.3 转向语句——goto

goto 语句又称转向语句，用来将程序无条件跳转到指定的标号语句处，其说明语句的一般形式如图 4-31 所示。

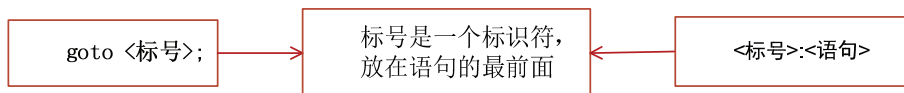


图 4-31 goto 语句的一般形式

【示例 4-16】程序中使用 goto 语句进行跳转。其实现代码及结果如图 4-32 所示。

```
#include <iostream.h>
int main()
{
    for(int i=0;i<5;i++)
    {
        if(i==2)
            goto END;
        cout<<i<<endl;
    }
    END:
    cout<<"hello"<<endl;
    return 0;
}
```

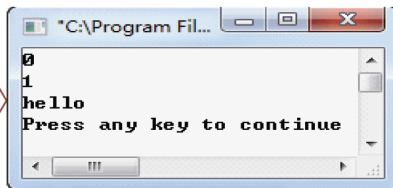


图 4-32 goto 语句



**注意：**使用 goto 语句将使程序结构不清晰，可读性低。一般来说，在结构化程序设计中应尽量少用或不用 goto 语句。



## 4.6 综合实例

本节将选择两个典型的综合实例，讲解如何在 C++ 中使用程序控制语句。希望读者在学习了这些例子后，能对 C++ 的基本语法有更详细的了解。

【示例 4-17】找出某个范围内的素数。所谓素数，就是只能被 1 和自身整除的正整数。比如，1 就是一个素数，2 和 3 也是，但 4 不是。因为 4 除了可以被 1 和自身整除外，还可以被 2 整除。从这个定义出发，要判断一个数是不是素数可以采用下面的算法：先判断这个数是不是 1 或 2，若是，则返回 true。如果不是，则求出该数的平方根，然后遍历大于等于 2 并小于这个平方根的所有整数，如果该数能够被这样的整数整除，则不是素数，否则就是一个素数。程序代码如下，运行结果如图 4-33 所示。

```
#include<iostream>
#include<cmath>
using namespace std;
int main()
{
    cout<<"-----求素数-----"<<endl;           //输出提示信息

    int min=0,max=0;                                   //素数的范围

    cout<<"请输入一个范围: "<<endl;                 //输出提示信息
    cout<<"大于等于:  ";
    cin>>min;                                           //输入最小值
    cout<<endl;

    if(min<1)                                           //求取范围只能是正整数
    {
        min=1;
    }

    cout<<"小于:  ";                                   //输入最大值
    cin>>max;
    cout<<endl;

    while(max<=min)                                     //如果输入有误，直到用户输入正确的最大值为止
    {
        cout<<"范围大上限必须大于下限。"           //输出出错信息
        <<"请重新输入:  "<<endl;
        cin>>max;                                       //重新输入最大值
    }

    cout<<"上述范围内的素数是:  "<<endl;

    for(int i=min;i<max;i++)                             //遍历范围内的所有整数
    {
        bool flag=true;                                  //表明当前整数是否是素数的标志
        if(1==i||2==i)                                   //如果 1 和 2 在范围内，直接输出
        {
            flag=true;
        }
        else                                             //除 1 和 2 外的其他数
        {
            int s=static_cast<int>(sqrt(i));             //求平方根
            for(int j=2;j<=s;j++)                         //从 2 开始遍历，直到平方根
            {
```





```

        if((i%j)==0)                //如果可以被整除，则不是素数
        {
            flag=false;
        }
    }

    if(flag)                        //若是素数，则输出
    {
        cout<<i<<endl;
    }
}

return 0;
}

```

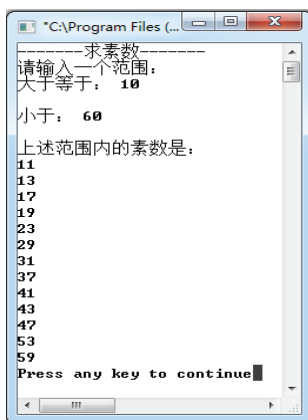


图 4-33 输出某范围内的素数

【示例 3-1】求最大值。用户输入 5 个整数，求其中的最大值。程序代码如下，程序的运行效果如图 4-34 所示。

```

#include<iostream>
#include<cmath>
using namespace std;
int main()
{
    cout<<"-----求最大值-----"<<endl;    //输出提示信息

    int temp=0;                                //保存输入值的变量
    int max=0;                                //保存最大值的变量
    cout<<"请输入 5 个整数: " <<endl;          //输出提示信息

    for(int i=0;i<5;i++)                      //循环 5 次
    {
        cin>>temp;                            //输入数据
        if(temp>max)                          //若当前输入的数据比 max 大
        {
            max=temp;                        //修改 max
        }
    }

    cout<<"最大值是: " <<max<<endl;          //输出最大值

    return 0;
}

```

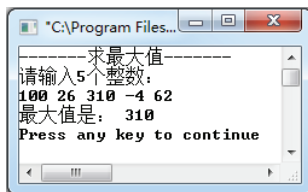


图 4-34 输出最大值

## 4.7 小结

本章主要介绍了 C++ 中实现程序控制结构的各种语句。针对每一种结构，本章都给出了各自的实现语句，并给出了示例供读者理解。在实际的应用中，这几种基本结构的使用非常频繁，读者应熟练掌握其实现语句和基本思想。最后，本章简要介绍了 C++ 中的 3 个转向语句，这些语句也可以控制程序的流程，但都有些使用限制，读者要仔细理解。

## 4.8 习题

【题目 4-1】某学校进行成绩考核。规定成绩在 60 分以上（含 60 分）的就是合格，60 以下的就是不合格。编写程序代码，实现判断某个人的成绩是否合格。若合格就输出“passed”，不合格就输出“failed”。要求：成绩数据从键盘输入，通过程序判断后，将结果输出到显示器上。运行效果如图 4-35 和图 4-36 所示。

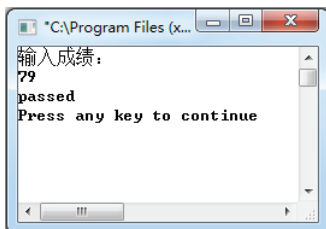


图 4-35 运行效果（1）

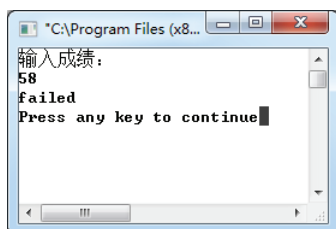


图 4-36 运行效果（2）

【题目分析】本题考查的是简单的条件选择。根据题目要求，只需要用一个 if 语句对成绩大于 60 分进行判断。若条件为真，则执行一条输出语句，输出合格信息；否则，转向执行另一条输出语句，输出不合格信息。

### 【关键代码】

```
if(grade>=60)
    cout<<"passed"<<endl;
else
    cout<<"failed"<<endl;
```

【题目 4-2】某学校进行成绩测试。规定成绩等级分为 A、B、C、D 四个等级。其中 90 分以上（含 90 分）的获得 A 级，90 分以下但在 80 分以上（含 80 分）的获得 B 级，80 分以下但在 60 分以上（含 60 分）的获得 C 级，60 分以下的获得 D 级。要求：

- （1）从键盘接收成绩数据，通过程序进行判断后，输出相应的等级。
- （2）分别用多重 if 语句和 switch 语句进行编程。



程序的运行效果如图 4-37~图 4-40 所示。

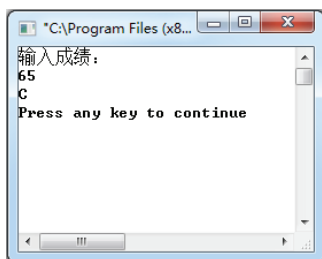


图 4-37 运行效果 (1)

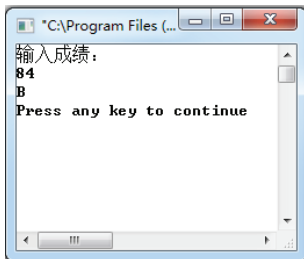


图 4-38 运行效果 (2)

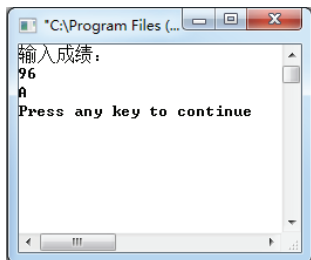


图 4-39 运行效果 (3)

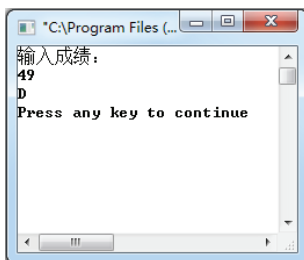


图 4-40 运行效果 (4)

**【题目分析】** 本题考查的知识较多，具有一定的综合性。要求读者能够将题目中给出的信息转换为条件判断，而且需要用逻辑运算符同关系运算符结合实现多条件的表达。另外，需要熟练掌握 if 语句和 switch 语句的用法。

**【关键代码】** 多重 if 语句编程：

```
cin>>grade;
if(grade>=90)
    cout<<"A"<<endl;
else if((grade<90)&&(grade>=80))
    cout<<"B"<<endl;
else if((grade<80)&&(grade>=60))
    cout<<"C"<<endl;
else
    cout<<"D"<<endl;
```

switch 语句编程：

```
cin>>grade;
switch(grade/10)
{
    case 0:
    case 1:
    case 2:
    case 3:
    case 4:
    case 5:
        cout<<"D"<<endl;break;
    case 6:
    case 7:
        cout<<"C"<<endl;break;
    case 8:
        cout<<"B"<<endl;break;
    case 9:
        cout<<"A"<<endl;break;
}
```



【题目 4-3】利用循环语句，求出 1~100 之间的奇数，效果如图 4-41 所示。

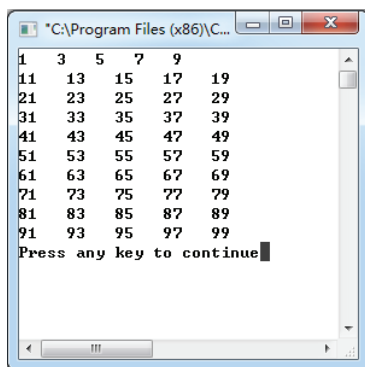


图 4-41 运行效果

【题目分析】上述题目主要是要求读者熟悉循环控制语句。另外，在 100 以内的奇数有 50 个，如果输出时不加控制，则结果阅读起来就会很困难。所以，读者还要考虑输出时数据如何组织。比如，当一行满 5 个数时输出一个换行，这样，输出结果就会很清晰。

【关键代码】

```
for (i=1; i<100; i++)
{
    if (i%2!=0)
    {
        cout<<i<<" ";
        m++;
    }
    if ( (m%5==0) && (m!=0) )
    {
        cout<<endl;
        m=0;
    }
}
```

【题目 4-4】编写一个程序，要求可以接收用户输入的数值，当数值大于 0 时，输出 “>0”；当数值小于 0 时，输出 “<0”；当数值等于 0 时，输出 “=0”。

【题目分析】要求读者熟悉关系运算符和选择控制语句。

【关键代码】

```
cout<<"please input number:";
cin>>num;
if (num==0)
{
    cout<<"输入的数值等于 0"<<endl;
}
else
{
    if (num<0)
    {
        cout<<"输入的数值小于 0"<<endl;
    }
    else
    {
        cout<<"输入的数值等于 0"<<endl;
    }
}
```



【题目 4-5】一个班级有 5 个人，他们分别为 A、B、C、D 和 E，对应学号分别为 1、2、3、4、5。现在设计程序，由用户输入学号，程序会给出对应姓名。要求：采用 switch 语句进行判断，程序不但能够根据用户的输入给出正确结果，还应该对输入的不合法数据具有一定的处理能力。程序的运行效果如图 4-42 所示。



图 4-42 运行效果

【题目分析】本题主要考查 switch 语句中的 default 分支语句。若开关表达式的值与所有的 case 语句中的常量表达式都不匹配，则执行 default 语句。

#### 【关键代码】

```
int x;
cout<<"请输入学号: "<<endl;
cin>>x;
switch(x)
{
case 1:
    cout<<"姓名: A"<<endl;
    break;
case 2:
    cout<<"姓名: B"<<endl;
    break;
case 3:
    cout<<"姓名: C"<<endl;
    break;
case 4:
    cout<<"姓名: D"<<endl;
    break;
case 5:
    cout<<"姓名: E"<<endl;
    break;
default:
    cout<<"请检查输入是否正确"<<endl;
}
```

【题目 4-6】下面是一个 C++ 程序代码，这个程序的运行结果如图 4-43 所示，请读者思考一下为什么输出这样的结果。

```
#include <iostream>
using namespace std;
int main( )
{
    int i;
    for(i=1; i<=7; i++)
        switch(i%3)
        {
            case 0: cout<<"Y"; break;
            case 1: cout<<"@"; break;
            case 2: cout<<"!";
                default: cout<<endl;
            }
    return 0;
}
```

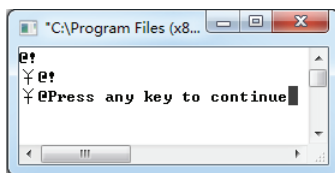


图 4-43 运行结果

【题目分析】本题考查的是 switch 语句的用法。在使用 switch 语句时，要注意将 case 分支语句与 break 语句结合使用，这样才能达成 switch 语句的真正目的。如果不加 break 语句，则在执行时，匹配的 case 语句后面的语句都会被执行，直到遇到 break 语句才会停止。所以，缺少 break 的语句不能正常跳出 switch 语句块。

【关键代码】

这两条语句后面都没有 break 语句。

```
case 2: cout<<"!";  
default: cout<<endl;
```

# 第5章 数组

前面章节中所涉及的变量和常量都是一个数据，而且数据与数据之间的关系是松散的。本章将介绍一种数据集合类型——数组。数组是指在程序设计中，为了处理方便，把具有相同类型的若干变量按有序的形式组织起来，并为其指定一个名称和长度。

## 5.1 数组概述

数组就是一组按照顺序排列在一起且类型相同的多个数据。每个数据称为数组元素。数组由数组名和下标两部分构成，如图 5-1 所示。



图 5-1 数组的形式

数组有两个属性，如图 5-2 所示。

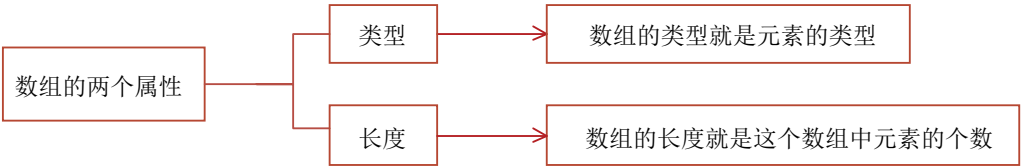


图 5-2 数组的两个属性

例如，一个存储 10 个整型数的数组，其类型就是整数，而长度就是 10，其内存布局如图 5-3 所示。

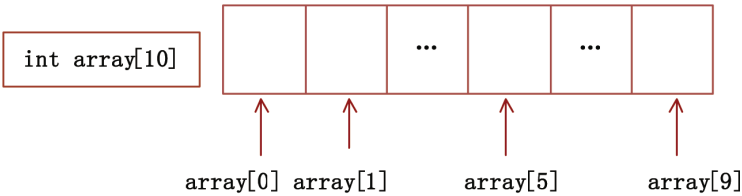


图 5-3 数组的内存布局



除了类型相同外，数组中的数据还有一个特点，即各元素在内存中按照顺序依次排列。也就是说，在内存中数组是一块连续的内存，其大小等于各元素所占内存之和。由于各元素类型相同，所以数组所占内存也等于元素个数与一个元素所占内存大小的乘积。在图 5-3 中，假设数组第一个元素的内存地址是 1000，则后续元素的地址依次递增一个整数所占字节数（本书采用的 32 位平台中整数占 4 字节），依次为 1004，1008，…直到 1036，如图 5-4 所示。

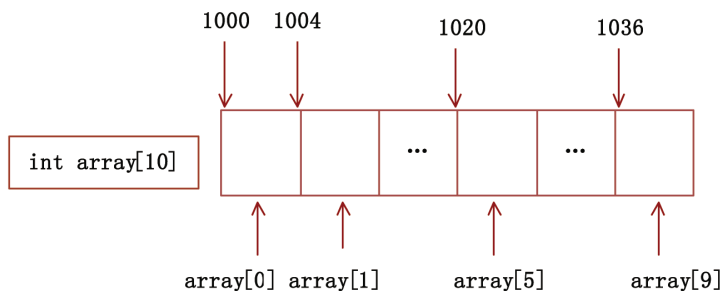


图 5-4 数组在内存中的地址



**注意：**不要把数组长度和数组所占内存大小混为一谈。数组长度是数组中所含元素的个数；数组所占的内存是其中各元素所占内存的总和。



## 5.2 数组的来源

前面章节中讲解的 C++ 的基本数据类型只能处理一些简单的数据类型，如果遇到复杂的具有相同类型的多个数据就很难解决。那么，C++ 提供了一种结构也就是数组来解决了一个问题。下面请看一个示例，从而让大家感受一下数组的优越性。

【示例 5-1】对一个班 40 人的期末成绩进行统计，用基本数据类型与一维数组的部分程序代码实现如图 5-5 所示。

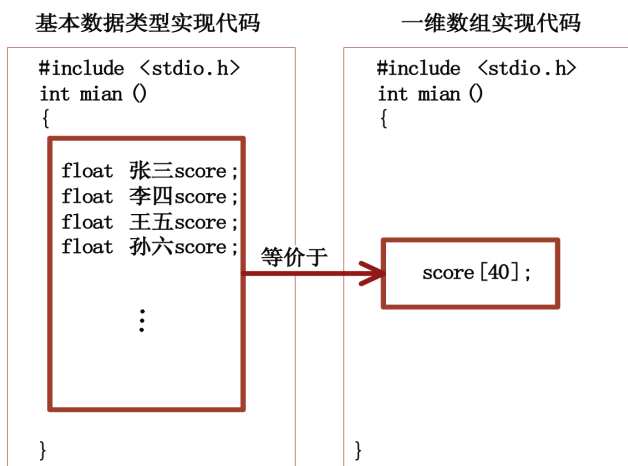


图 5-5 基本数据类型与一维数组比较





从这个例子中可以看出,对于变量的存储,使用数组的形式会比使用基本类型的存储方便,程序的可读性也会比较好。数组根据下标的个数,分为一维数组和多维数组。本章重点讲解一维数组和二维数组。



## 5.3 一维数组

一维数组在具体程序中使用是非常广泛的。一维数组是下标的个数只有一个的数组,即声明数组时只有一个[]包含的下标,如上面提到的数组 `array[10]` 就是一个一维数组。一维数组是最简单的一类数组。本节详细介绍一维数组的声明、定义、初始化和引用。

### 5.3.1 一维数组的声明和定义

要使用数组就要先声明和定义。一维数组的一般定义语法如图 5-6 所示。

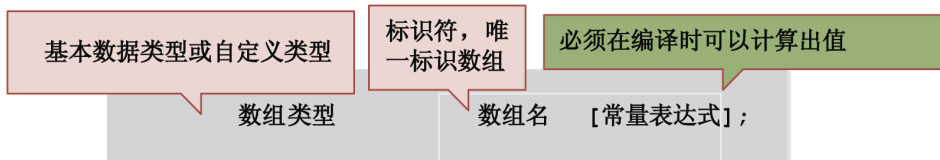


图 5-6 一维数组的声明和定义

编译器在接到声明数组的指令后,将开辟一个与数组大小相同的、连续存储的内存空间来存放数组中的元素,并用数组名和这块内存区域相关联。

自定义类型,如结构体、枚举体、类等,将在后面的章节中介绍。如图 5-7 所示定义了一个简单的数组。

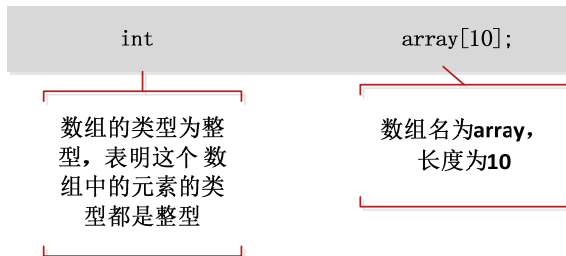


图 5-7 整型数组定义

下面几个数组的定义包含两个容易犯的错误,如图 5-8 所示。

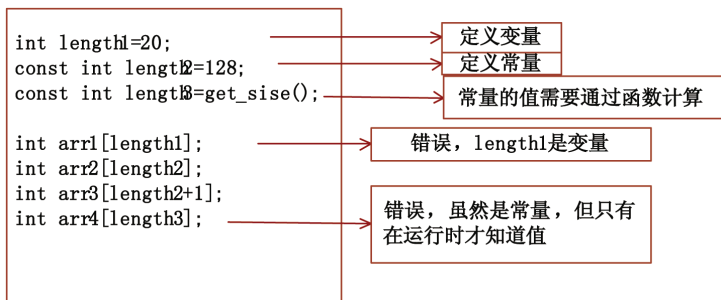


图 5-8 数组定义时容易犯的错误



同变量的定义一样，数组也可以连续定义，例如：

```
int a[30], b[20];
```

如果数组中的元素是不定的，则可以这样定义：

```
const int NUMBER=20;  
int a[NUMBER];
```

也可以用 `typedef` 来定义：

```
const int SIZE=50;  
typedef double list[SIZE];  
list yourList;  
list myList;
```

初始化数组就是在定义数组时，可以为每个数组元素提供初值。初始化数组的一般形式如图 5-9 所示。

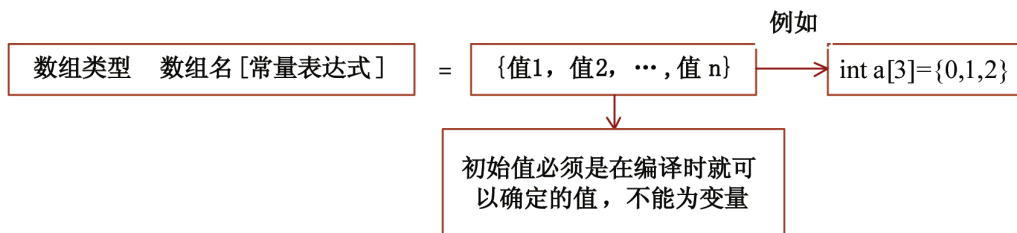


图 5-9 初始化数组

对于数组如果不提供初始化列表，则其值全部为随机数（不一定为 0）；如果提供初始化列表，但初始化列表中元素的个数少于数组的长度，则不足部分的数组元素用 0 来初始化。示例如图 5-10 所示。



图 5-10 列表中元素个数少于数组长度

但是初始化列表中元素的个数不能多于数组的长度，图 5-11 中数组的初始化是错误的。



图 5-11 列表中元素多于数组长度

如果在定义数组时为所有元素都提供了初始化值，那么代表数组长度的常量表达式可以省略。此时数组的长度就是初始值的个数。示例如图 5-12 所示。



图 5-12 所有元素提供了初始化

### 5.3.2 一维数组的引用

一维数组的数组元素引用的一般形式如图 5-13 所示。



数组中每个  
元素的序号

数组名 [下标]

图 5-13 一维数组引用的一般形式



**注意：**使用下标访问数组元素时，注意不可超出下标的取值范围。

在定义数组之后，不可以直接对数组进行赋值。要想修改其中元素的值，只能通过下标的方法先找到要操作的元素，然后进行处理。示例如图 5-14 所示。

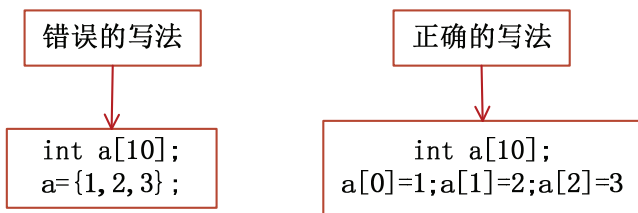


图 5-14 一维数组引用

实际中常用循环来改变数组元素的值，示例如图 5-15 所示。

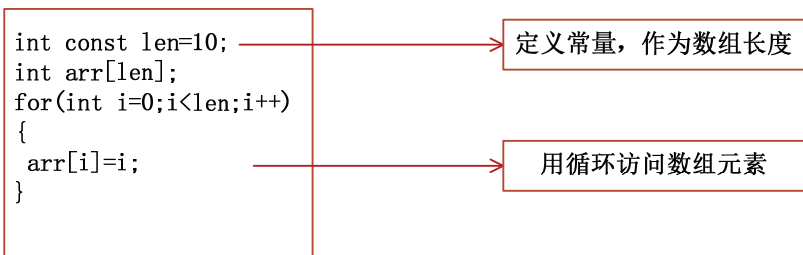


图 5-15 循环改变数组元素的值

学习 C++ 时，经常犯的一个错误就是直接对数组进行赋值等操作，例如，想要让两个数组的元素有相同的值，其操作如图 5-16 所示。

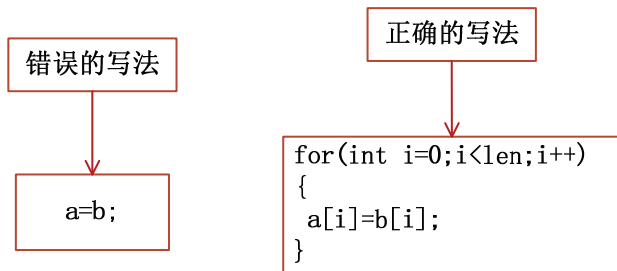


图 5-16 两个数组的元素值相同

**【示例 5-2】**从键盘接收 5 个数字，并将其按照输入顺序倒序输出。其实现代码及结果如图 5-17 所示。

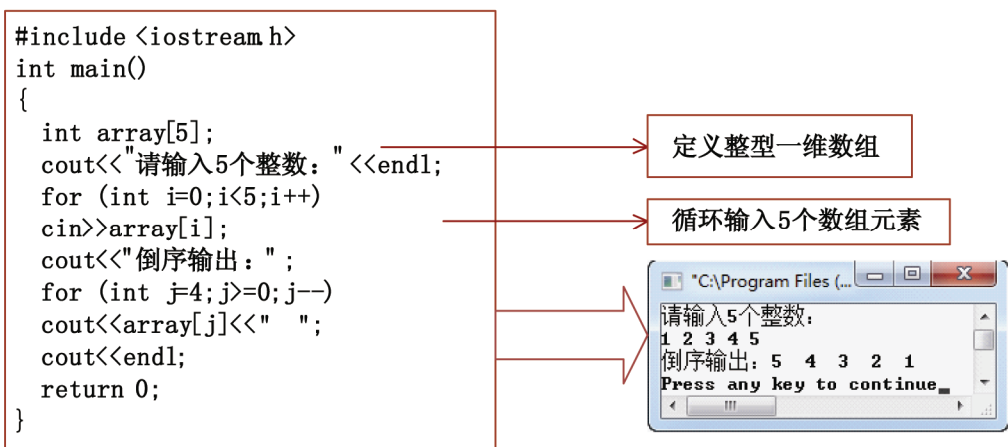


图 5-17 一维数组实例



**注意：**在一些新的编译器中，编写的程序中若有数组下标越界，程序将结束并返回一个错误信息。



## 5.4 二维数组

二维数组是多维数组在实际应用中最广泛的一种。二维数组是在一维数组声明方式的基础上，增加下标的维数，即增加“[]”的个数。二维数组就是声明数组时“[]”的个数为2，即有两个下标，可以分别表示行数和列数。本节将详细介绍二维数组的声明和定义、初始化和引用。

### 5.4.1 二维数组的声明和定义

二维数组声明和定义的一般形式如图 5-18 所示。

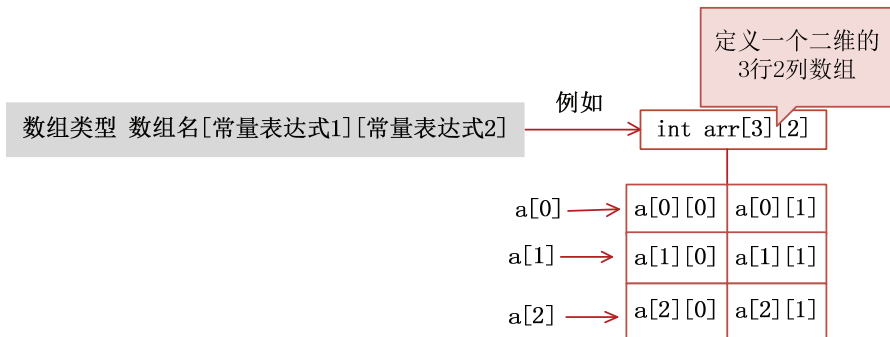


图 5-18 二维数组的声明和定义

### 5.4.2 二维数组的初始化

二维数组的初始化与一维数组类似，同样是在声明数组时加上初值表。其初始化的一般形式如图 5-19 所示。

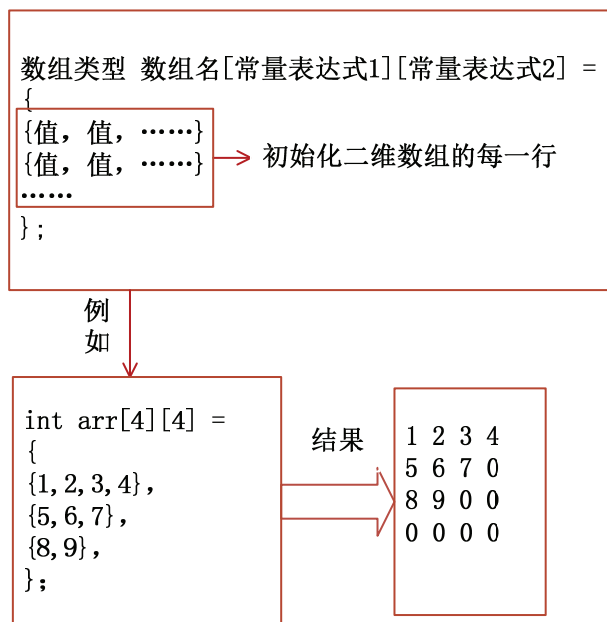


图 5-19 二维数组初始化的一般形式

对于二维数组的初始化, 如果前面的每一段都提供了足够的初始化值, 则花括号“{”可以省略。其初始化的形式如图 5-20 所示。

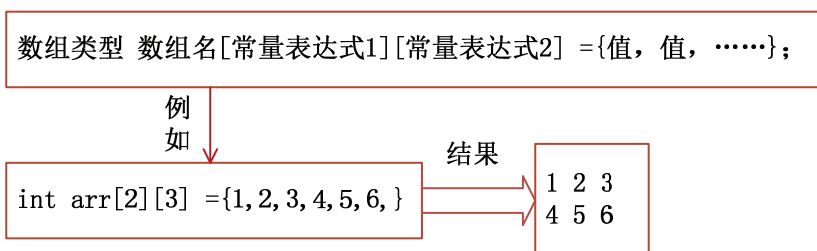


图 5-20 二维数组的初始化形式

这两种方法相比, 分段初始化更清晰, 也更灵活, 因为每一段都可以只提供部分初始化值。不过, 读者可根据实际情况进行选择。在二维数组初始化时, 如果为所有的元素都提供了初始化值, 则第一维的长度可以省略, 如图 5-21 所示。



图 5-21 第一维长度省略



**注意:** 对于二维数组, 只能省略第一维的长度, 而第二维的长度不能省略。



### 5.4.3 二维数组的引用

二维数组同一维数组一样，数组元素共用同一个名称，使用下标对每一个元素进行访问。不过，对于二维数组元素来说，要使用两个下标来访问。引用二维数组的一般形式如图 5-22 所示。

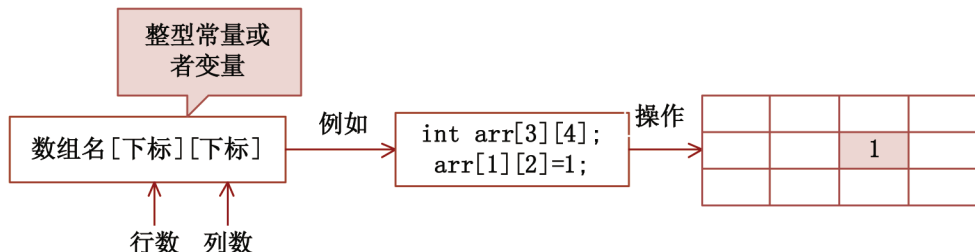


图 5-22 二维数组的引用

【示例 5-3】一个学习小组有 3 个人，每个人有 3 门课的考试成绩。求全组分科的平均成绩，其中成绩由用户从键盘输入。其实现代码及结果如图 5-23 所示。

```
#include <iostream.h>
int main()
{
    int stu[3][3], gra[3];
    int i, j, sum;

    cout<<"请输入学生成绩："<<endl;
    for (i=0; i<3; i++)
    {
        for(j=0; j<3; j++)
        {
            cin>>stu[i][j];
        }
    }

    cout<<"三门课程的平均分分别为：" <<endl;
    for(j=0; j<3; j++)
    {
        sum=0;
        for(i=0; i<3; i++)
        {
            sum+=stu[i][j];
        }
        gra[j]=sum/3;
    }

    for(j=0; j<3; j++)
    {
        cout<<gra[j]<<endl;
    }
    return 0;
}
```

定义二维数组 `stu[3][3]` 存放 3 个人 3 门课的成绩  
定义一维数组 `gra[3]` 存放各分科平均成绩

计算三门课程  
的平均成绩

图 5-23 二维数组实例



#### 5.4.4 多维数组在内存中如何排列元素

维数决定了数组中元素的组织方式及访问元素所用的下标个数，从本质上讲，所有的数组在内存中都是一维线性的。以二维数组为例，内存中是先放第一行的元素，再放第二行的元素，以此类推。图 5-24 给出了大小为  $3 \times 4$  的二维数组 A 的排列顺序。

```
A[0][0]-> A[0][1]-> A[0][2]-> A[0][3]->  
  
A[1][0]-> A[1][1]-> A[1][2]-> A[1][3]->  
  
A[2][0]-> A[2][1]-> A[2][2]-> A[2][3]
```

图 5-24 二维数组内存的组织

多维数组的存储方式与此类似，最左边一维下标的变化是最慢的，最右边一维下标变化最快。图 5-25 给出了  $3 \times 3 \times 3$  的三维数组 B 中元素在内存中的排列顺序。

```
B[0][0][0]-> B[0][0][1]-> B[0][0][2]->  
B[0][1][0]-> B[0][1][1]-> B[0][1][2]->  
B[0][2][0]-> B[0][2][1]-> B[0][2][2]->  
B[1][0][0]-> B[1][0][1]-> B[1][0][2]->  
B[1][1][0]-> B[1][1][1]-> B[1][1][2]->  
B[1][2][0]-> B[1][2][1]-> B[1][2][2]->  
B[2][0][0]-> B[2][0][1]-> B[2][0][2]->  
B[2][1][0]-> B[2][1][1]-> B[2][1][2]->  
B[2][2][0]-> B[2][02][1]-> B[2][2][2]->
```

图 5-25 三维数组内存的组织

由于多维数组在内存中是线性存储的，元素存放的顺序也是确定的。



## 5.5 字符串

在计算机中经常会处理字符，C++对字符的处理主要是通过字符数组实现的。本节将详细讲述字符数组、字符串及其之间的区别。

### 5.5.1 字符数组

字符数组是用来存放字符的数组。定义字符数组的一般形式如图 5-26 所示。

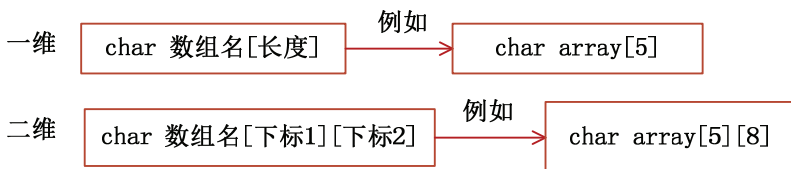


图 5-26 定义字符数组

同样，字符数组在声明的同时可以为其赋初值，其方法与数值数组相似，如图 5-27 所示。

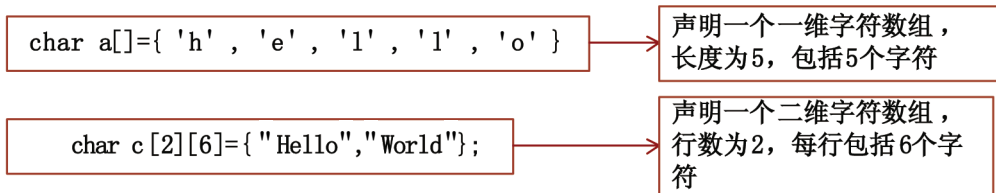


图 5-27 字符数组的初始化

此外，字符数组在内存中的存储与一般数值数组相同。一维字符数组在内存中按顺序存储，二维字符数组则一般按行存储。在字符数组元素的引用上，与一般数值数组也是类似的，只允许通过数组下标单个引用其中的字符数组元素。

【示例 5-4】下面程序接收用户不超过 10 个字符的输入，并将输入的字符倒序输出。其实现代码及结果如图 5-28 所示。

```
#include <iostream.h>
int main()
{
    char a[10];
    int i, j;
    cout<<"请输入不超过10个字符:" ;
    for(i=0; i<10; i++)
    {
        cin>>a[i];
    }
    cout<<"倒序输出: " <<endl;
    for(j=i-1; j>=0; j--)
    {
        cout<<a[j]<<" ";
    }
    cout<<endl;
    return 0;
}
```

定义一维字符数组

输入10个字符  
后退出循环

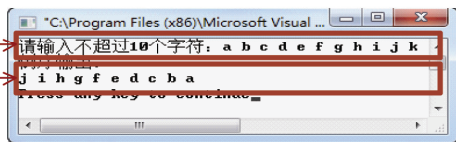


图 5-28 字符数组实例





### 5.5.2 字符串的存储形式

字符串是特殊的字符数组。与普通字符数组不同的是，字符串在结尾处有一个字符`\0`，表示字符串的结束。字符串用的是双引号`" "`，如`"Hello World"`表示一个字符串，其存储形式如图 5-29 所示。



图 5-29 字符串的存储形式



**说明：**字符串的结束标志不需要显式地提供，编译器会自动在后面追加。

所示字符串的优势主要体现在其输入、输出和赋初值上。例如，图 5-27 中 `a[]` 用字符串的形式可以有如图 5-30 所示的两种表示方式。

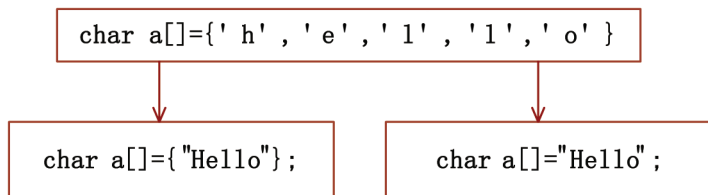


图 5-30 字符串的表示形式

可以看到，使用字符串的形式可以简化赋初值，同时在进行输入/输出时可以不需要使用循环语句来完成。

【示例 5-5】下面程序不使用循环语句直接输入/输出一个字符串。其实现代码及结果如图 5-31 所示。

```
#include <iostream.h>
int main()
{
    char str[10];
    cout<<"请输入字符串:";
    cin>>str;
    cout<<"输出字符串为:";
    cout<<str<<endl;
    return 0;
}
```

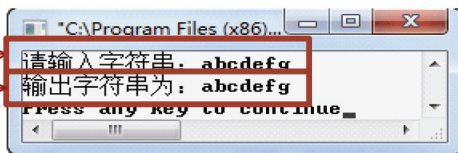


图 5-32 字符串输入/输出实例

### 5.5.3 字符数组与字符串的区别

在 C++ 中，字符数组和字符串的声明都是通过 `char` 关键字来定义的，但两者是不同的。两者最显著的区别在于，字符串的长度是其中字符的数目再加 1，因为其包含了结束符`\0`，而字符数组的长度就是其中字符的数目。

【示例 5-6】下面程序定义一个字符串 `a` 和一个字符数组 `b`，分别计算其长度，可看出字符串和字符数组的区别。实现代码及结果如图 5-33 所示。

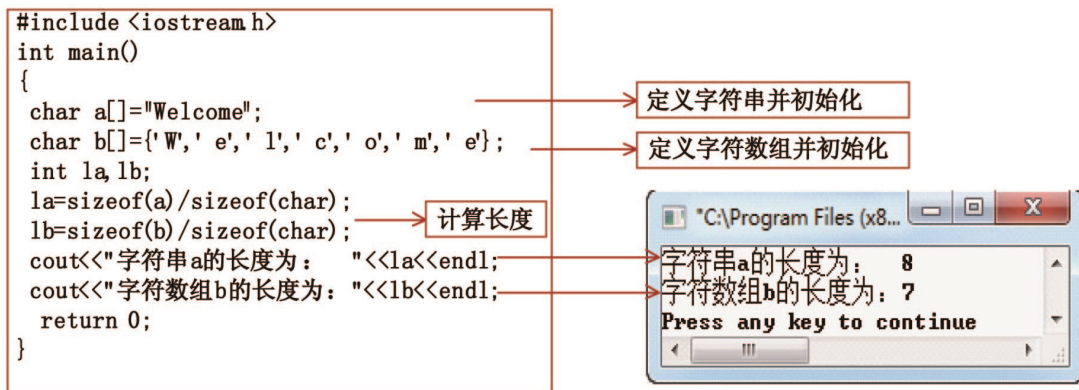


图 5-33 字符数组与字符串的区别

## 5.5.4 字符串处理函数

字符串可以使用 `cout` 和 `cin` 进行整体的输入/输出，但是，其他整体操作，如赋值、比较和连接等都是不允许的，如图 5-34 所示。

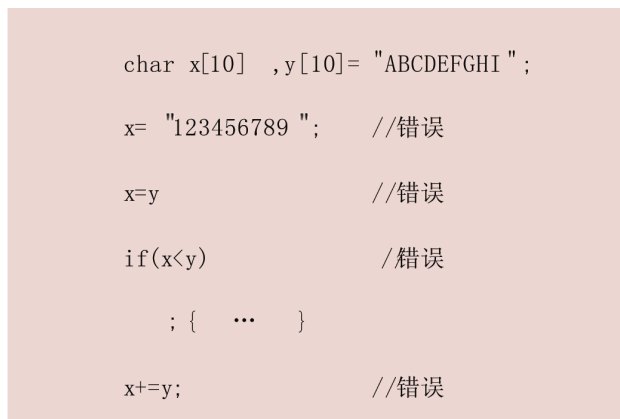


图 5-34 字符串整体操作

和普通数组一样，原则上，这些操作都必须逐个对字符串中的元素进行操作，但为了方便程序员对字符串进行处理，系统提供了相应的库函数来完成这些操作，标准头文件 `cstring`（旧标准为 `string.h`）提供了很多相关函数的声明，常用的字符串处理函数如表 5-1 所示。

表 5-1 常用字符串处理函数

操作	函数原型	备注
取得字符串的长度	<code>size_t strlen(数组名)</code>	不包括空字符
复制字符串	<code>char* strcpy(目标数组名, 源数组名)</code>	目标数组元素个数应不小于源数组中的元素个数
字符串相等比较	<code>int strcmp(数组名 1, 数组名 2)</code>	数组 1 和数组 2 相等则返回 0，如果数组 1 小于数组 2，返回一个负数，否则，返回一个正数
将小写字母都转换成大写字母	<code>char* strpr(数组名)</code>	
连接两个字符串	<code>char* strcat(数组名 1, 数组名 2)</code>	在字符串组 1 后接上字符串组 2



**说明：**函数中的数组，指的都是最后一个元素是空字符的字符数组。字符串比较大，实际就是比较字符串的 ASCII 码。比较规则：将两个字符串中的字符，从左到右依次比较，根据当前字符特性将字符按照字典顺序进行逐一比较。字典排序靠前的字符小，遇到不相等的字符就按这个位置上的两个字符的比较结果确定两个字符串的大小。

下面，来举一个例子说明字符串处理函数的用法，如图 5-35 所示。

```
#include <iostream>
#include <cstring>
int main()
{
    using namespace std;
    char password[]="CHINA";
    char input[6],copyInput[6];
    char catInput[]="I LOVE ";
    cout<<"请输入密码（不超过5位）："
        <<endl;
    cin>>input;
    cout<<"你输入的密码位数为："
        <<strlen(input)<<endl;
        //长度计算
    cout<<"转换成大写形式 "
        <<strupr(input)<<endl;
        //全部转换成大写形式
    if(strcmp(strupr(input),password)==0)
        //比较两个字符串是否相等
        cout<<"密码正确"<<endl;
    strcpy(copyInput,input); //复制函数
    cout<<"执行字符串复制操作后："
        <<copyInput<<endl;
    strcat(catInput,input);
    //拼接函数，将input接在catInput后
    cout<<"执行字符串连接操作后："
        <<catInput<<endl;
    return 0;
}
```

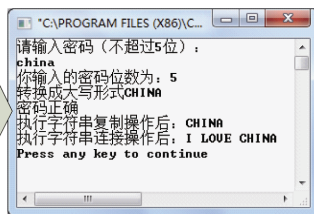


图 5-35 字符串处理函数的使用

上例演示了如何使用库函数来处理字符串，`cstring` 头文件中有更多的处理函数的声明，在需要对字符串进行操作时应尽量使用库函数。



## 5.6 综合实例——杨辉三角

数组和字符串在实际程序中的应用非常广泛。为了使读者更好地理解数组和字符串的应用，下面给出一个较为综合的示例，读者可仔细理解数组在其中的作用。

【示例 5-7】下面程序从键盘接收用户输入的一个不超过 10 的整数，输出该整数行的杨辉三角。其实现代码及结果如图 5-36 所示。

概念	程序代码	注释
<p>杨辉三角是一个特征数，其每一行的开头和结尾数字都是1，中间所有数字都可以由前一行前后两个数相加得到</p>	<pre>#include &lt;iostream h&gt; int main() {     int n, i, j;     cout&lt;&lt;"请输入行数:" ;     cin&gt;&gt;n;     int a[20]={0}, b[20]={0};     for( i = 0; i &lt; n; i ++ )     {         b[0] = 1;         b[i] = 1;         for( j = 1; j &lt; i; j ++ )         {             b[j] = a[j] + a[j-1];         }         for( j = 0; j &lt; n - i - 1; j ++ )             cout&lt;&lt;' ' ;         for( j = 0; j &lt;=i; j ++ )         {             if( j &gt; 0 ) cout&lt;&lt;' ' ;             cout&lt;&lt;b[j];         }         cout&lt;&lt;endl;         for( j = 0; j &lt;=i; j++ )             a[j] = b[j];     }     return 0; }</pre>	
<p>示意图</p>		
<p>结果图</p>		

图 5-36 杨辉三角

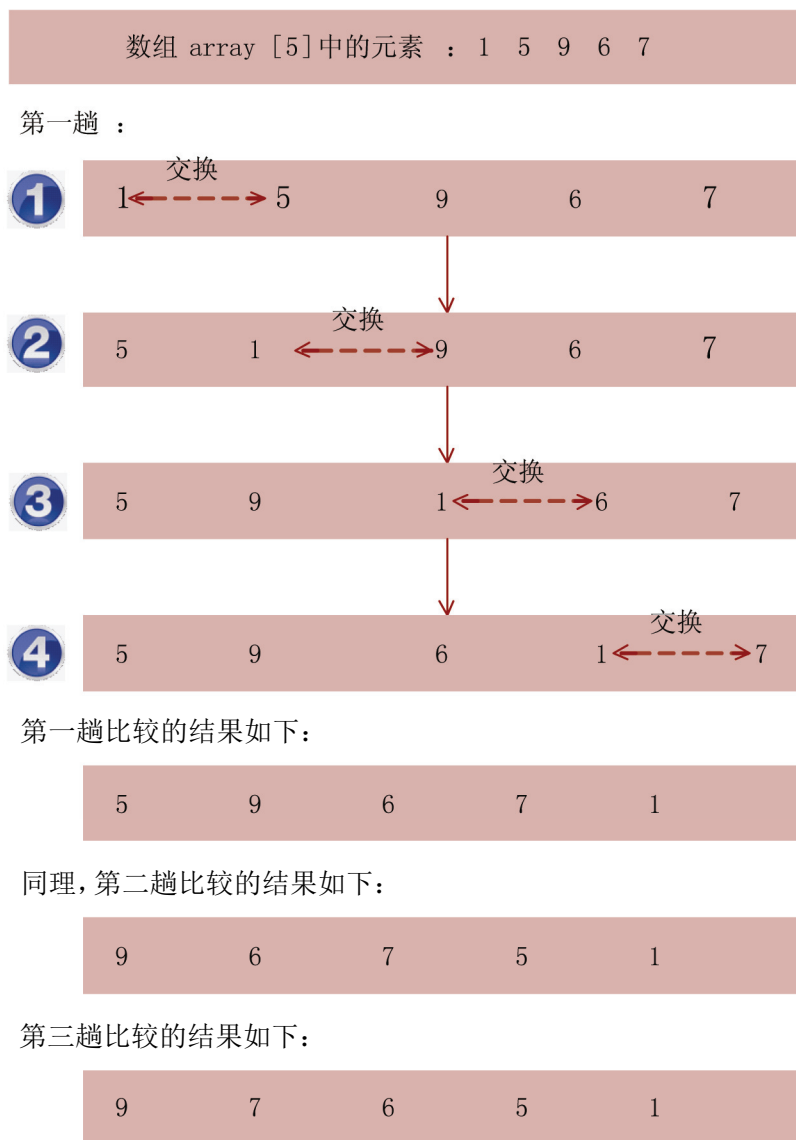
【示例 5-8】数组元素排序。在这个示例程序中，需要输入 10 个数，按照从小到大的顺序重新排列，然后输出到屏幕，这里重点展示怎么使用数组，算法采用冒泡排序。

冒泡排序的基本概念：依次比较相邻的两个数，将大数放在前面，小数放在后面。即首先比较第 1 个和第 2 个数，将大数放在前面，小数放在后面。然后比第 2 个和第 3 个数，同样大数在前小数在后。如此继续，直至比较最后两个数。此时第一趟排序结束，在最后面的数必定是所有数中的最小数。重复以上过程，第二趟结束，在倒数第二个位置上的就是新的最小数。



如此，直至完成排序。

由于在排序过程中总是大数往前放，小数往后放，相当于气泡往上升，所以称为“冒泡排序”。图 5-37 用一个简单的例子演示该算法的工作过程。



此时，排序目的达到，以后的比较中将不再交换

图 5-37 冒泡排序算法

程序源代码如图 5-38 所示，执行效果如图 5-39 所示。



```
#include <iostream>
using namespace std;
int main( )
{
    int const len=10;
    int arr[len];
    cout<<"请输入10个元素: "<<endl;
    int i;
    for(i=0;i<len;i++)
    {
        cin>>arr[i];
    }
    int j;
    for(i=0;i<len;i++)
    {
        for(j=0;j<len-i-1;j++)
        {
            if(arr[j]<arr[j+1])
            {
                int temp=arr[j];
                arr[j]=arr[j+1];
                arr[j+1]=temp;
            }
        }
    }
    cout<<"输入的数据从大到小的排序
    结果为: "<<endl;
    for(i=0;i<len;i++)
    {
        cout<<arr[i]<<" ";
    }
    cout<<endl;
    return 0;
}
```

依次输入数组元素

这个双重循环是本程序的关键，外层循环控制排序的趟数，内层循环保证最小的数浮到后面

输出数组元素

图 5-38 冒泡排序法

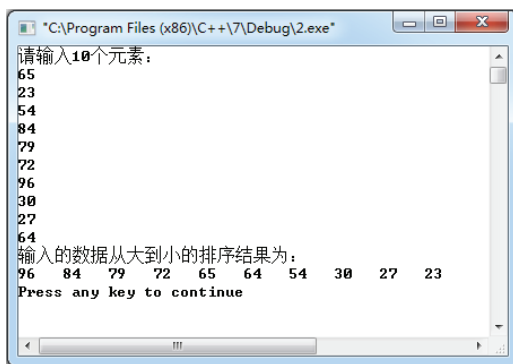


图 5-39 运行结果

在上述代码中，数组长度为 10，其有效下标范围为 0~9。代码中的重点是双层循环，外层循环控制排序的趟数，内层循环保证最小的数浮到后面。



## 5.7 小结

本章主要介绍了 C++ 中非常重要的一种组合型数据类型——数组。本章对于数值型一维数组和二维数组的声明、初始化和引用等内容做了重点讲解。对字符数组、字符串等内容也做了详细介绍。最后通过一个综合实例进一步掌握数组的应用。

## 5.8 习题

【题目 5-1】采用一维数组存储 10 个学生的成绩，然后计算这 10 个学生的平均成绩。要求：一维数组中存放的成绩数据由键盘输入，成绩作为结果输出到显示器上。运行效果如图 5-42 所示。

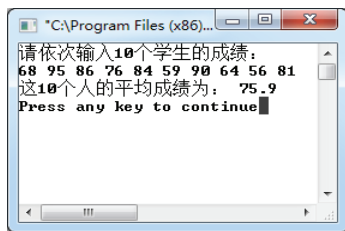


图 5-42 运行效果

【题目分析】本题考查一维数组的声明和数组元素的引用。

【关键代码】

```
int grade[10];
int sum=0;
double aver;
cout<<"请依次输入 10 个学生的成绩: "<<endl;
for(int i=0;i<10;i++)
{
    cin>>grade[i];
    sum+=grade[i];
}
aver=sum/10.0;
cout<<"这 10 个人的平均成绩为: "<<aver<<endl;
```

【题目 5-2】计算机系的同学李强，在学习了 C++ 程序设计这门课的数组一章后，编写了一个简单的程序。程序中定义了几个一维数组，他想把一维数组中的值输出。但是，由于数组定义时出现了问题，不能输出结果。李强同学很着急，请你帮助他解决问题，改正数组定义中的错误，使程序能将数组中存放的数据进行输出。程序的正确运行效果如图 5-43 所示。

```
#include<iostream>
using namespace std;
int main()
{
    int a=5;
    int array1[4]={a,a+1,a+2,a+3},array2[5],array3[a]={1,2,3,4,5 };
    array2=array3;
    cout<<array1<<endl;
    cout<<array2<<endl;
    cout<<array3<<endl;
    return 0;
}
```

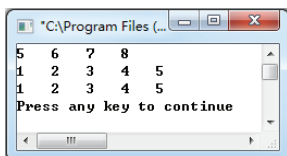


图 5-43 正确的运行效果

【题目分析】本题考查一维数组的定义和初始化。在定义数组时，数组的长度不能是变量，必须是在编译时就能确定的值。在引用数组中的元素时不能进行整体的引用，应逐个元素进行输入、输出及赋值等操作，这时可以用循环来进行赋值和输出。这是在定义和引用数组时最容易犯的错误，其次，对数组中的数据进行访问时，尤其要注意数组越界问题。

#### 【关键代码】

```
int i=0;
int array1[4]={5,6,7,8},array2[5],array3[5]={1,2,3,4,5};
for(i=0;i<5;i++)
{
    array2[i]=array3[i];
}
for(i=0;i<4;i++)
{
    cout<<array1[i]<<" ";
}
cout<<endl;
for(i=0;i<5;i++)
{
    cout<<array2[i]<<" ";
}
cout<<endl;
for(i=0;i<5;i++)
{
    cout<<array3[i]<<" ";
}
cout<<endl;
```

【题目 5-3】有 10 个程序员参加笔试，满分为 100 分，分为 5 等。这 5 等分别是 A 等（90 以上含 90 分），B 等（80~90 含 80 分），C 等（70~80 含 70 分），D 等（60~70 含 60 分），E 等（60 分一下）。现在需要统计这次考试成绩中 5 个分数段的人数，并计算他们的平均分，以及他们中的最高分是多少。要求：这 10 个程序员的成绩从键盘输入，并把输入的数据保存在一维数组中。程序的运行效果如图 5-44 所示。

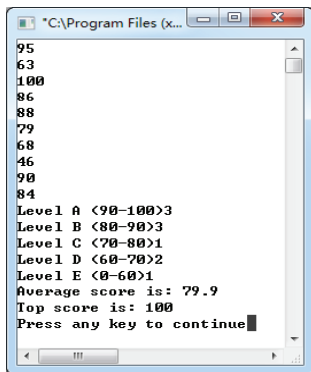


图 5-44 运行效果





【题目分析】本题考查的是一维数组的应用问题。注意：在对数组元素操作时要逐个使用，不能整体使用。

【关键代码】

```
double score[10];
int A=0,B=0,C=0,D=0,E=0,i=0;
double topscore=0;
double totalscore=0;
for(i=0;i<10;i++)
{
    cin>>score[i];
}
for(i=0;i<10;i++)
{
    switch ((int)(score[i]/10))
    {
        case 10:
        case 9:A++;break;
        case 8:B++;break;
        case 7:C++;break;
        case 6:D++;break;
        case 5:
        case 4:
        case 3:
        case 2:
        case 1:
        case 0:E++;break;
    }
}

cout<<"Level A (90-100)"<<A<<endl;
cout<<"Level B (80-90)"<<B<<endl;
cout<<"Level C (70-80)"<<C<<endl;
cout<<"Level D (60-70)"<<D<<endl;
cout<<"Level E (0-60)"<<E<<endl;

for(i=0;i<10;i++)
{
    totalscore+=score[i];
}

cout<<"Average score is: "<<totalscore/10<<endl;

topscore=score[0];
for(i=0;i<10;i++)
{
    if(topscore<score[i])
    {
        topscore=score[i];
    }
}

cout<<"Top score is: "<<topscore<<endl;
```

【题目 5-4】编写程序将字符串反转。要求：从键盘输入一个字符串，经过程序处理后，将反转后的字符串输出到显示器上。运行效果如图 5-45 所示。

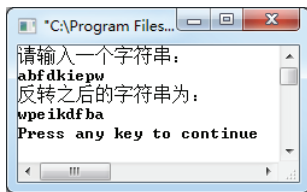


图 5-45 运行效果

【题目分析】本题考查字符串和字符数组的知识。题目中，从键盘输入的字符串需要保存在一个字符数组中。然后，通过数组对字符串进行反转操作。反转的算法是：第一个字符和最后一个字符交换，第二个字符和倒数第二个字符交换，依次类推。假定输入的字符串为 abcdefg，那么其交换过程示意如图 5-46 所示。

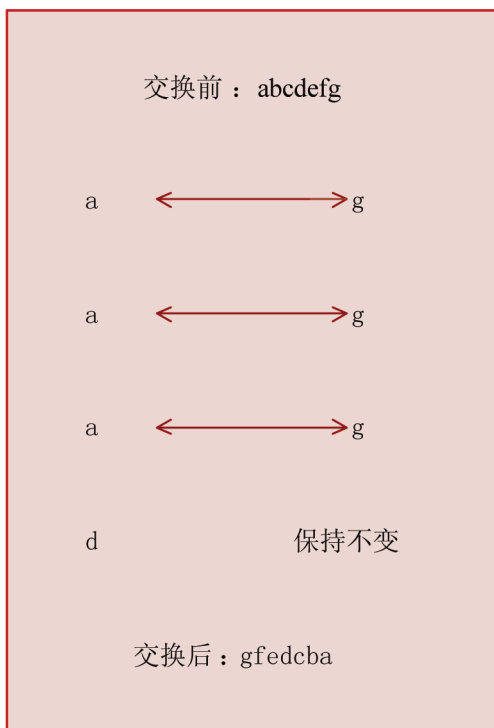


图 5-46 字符串反转

### 【关键代码】

```
char buffer[128];
cout<<"请输入一个字符串: "<<endl;
cin>>buffer;
int len=(int)strlen(buffer);
for(int i=0;i<len/2;i++)
{
    char temp=buffer[i];
    buffer[i]=buffer[len-i-1];
    buffer[len-i-1]=temp;
}
cout <<"反转之后的字符串为: "
<<endl<<buffer<<endl;
```



【题目 5-5】将两个字符串的内容互换。要求：编写程序，接收用户从键盘输入的两个字符串，经过程序处理后，将互换后的字符串输出到显示器上，如图 5-47 所示。程序的运行效果如图 5-48 所示。

【题目分析】考查字符串交换函数 `strcpy` 的使用。建立两个字符数组分别接收来自键盘输入的字符串，再建立一个字符数组作为中间数组。交换两个数组的算法与交换两个变量类似，只是交换时要用到字符串复制函数。

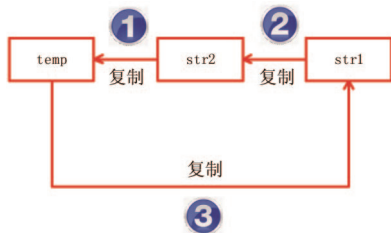


图 5-47 字符串的交换

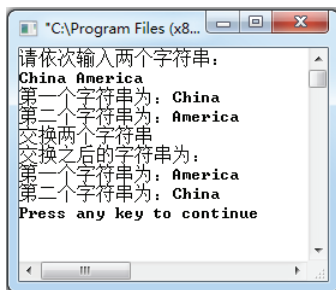


图 5-48 运行效果

#### 【关键代码】

```

char str1[128], str2[128];
cout<<"请依次输入两个字符串: "<<endl;
cin>> str1;
cin>> str2;

cout<<"第一个字符串为: "<<str1<<endl;
cout<<"第二个字符串为: "<<str2<<endl;

cout<<"交换两个字符串"<< endl;
char temp[128];
strcpy(temp, str1);
strcpy(str1, str2);
strcpy(str2, temp);

cout<<"交换之后的字符串为: "<<endl;
cout<<"第一个字符串为: "<<str1<<endl;
cout<<"第二个字符串为: "<<str2<<endl;
  
```

【题目 5-6】将两个字符串连接起来。要求：编写程序，接收用户从键盘输入的两个字符串，经过程序处理后，将连接好的字符串输出到显示器上。程序的运行效果如图 5-49 所示。

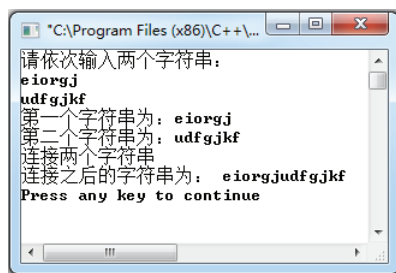


图 5-49 运行效果

【题目分析】本题考查字符串的连接函数 `strcat(string1,string2)`。字符串连接函数将连接好的字符串放在 `string1` 中。

【关键代码】

```
char str1[128], str2[128];\ncout<<"请依次输入两个字符串: "<<endl;\ncin>> str1;\ncin>> str2;\n\ncout<<"第一个字符串为: "<<str1<<endl;\ncout<<"第二个字符串为: "<<str2<<endl;\n\ncout<<"连接两个字符串"<< endl;\n\nstrcat(str1,str2);\ncout<<"连接之后的字符串为: "<<str1<<endl;
```

【题目 5-7】比较两个字符串的大小。要求：编写程序，接收用户从键盘输入的两个字符串，经过程序处理后，将比较的结果输出到显示器上。程序的运行效果如图 5-50 所示。

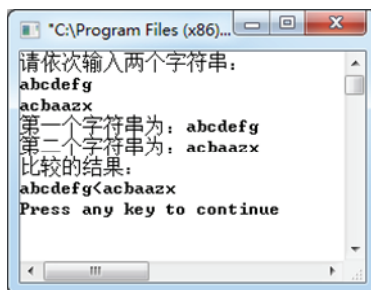
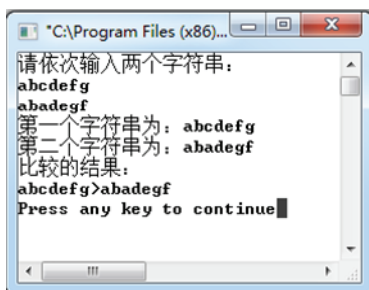
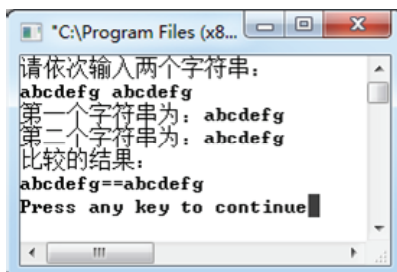


图 5-50 运行效果

【题目分析】本题考查两个字符串的比较函数 `strcmp (string1,string2)`。字符串的比较规则是按 ASCII 码顺序比较两个字符串，并由函数的返回值判断结果。当 `string1` 等于 `string2`



时, 返回值为 0; 当 string1 小于 string2 时, 返回值小于 0; 当 string1 大于 string2 时, 返回值大于 0。

### 【关键代码】

```
cout<<"请依次输入两个字符串: "<<endl;
cin>> str1;
cin>> str2;

cout<<"第一个字符串为: "<<str1<<endl;
cout<<"第二个字符串为: "<<str2<<endl;

cout<<"比较的结果: "<<endl;
int result=strcmp(str1,str2);
cout<<str1;
if(result==0) cout<<"==";
if(result>0) cout<<">";
if(result<0) cout<<"<";
cout<<str2<<endl;
```

【题目 5-8】转置一个二维数组。要求: 声明一个二维数组, 二维数组中存放的数据由用户从键盘输入。通过程序对二维数组进行处理后, 输出转置后的二维数组。程序的运行效果如图 5-51 所示。

【题目分析】本题主要考查二维数组的相关知识。所谓转置, 指的是将数组中的元素关于对角线转换, 如图 5-52 所示。

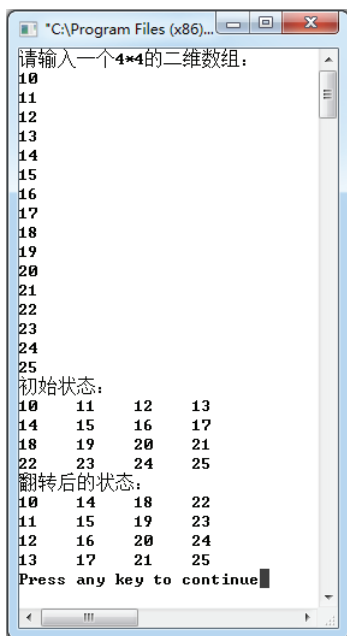
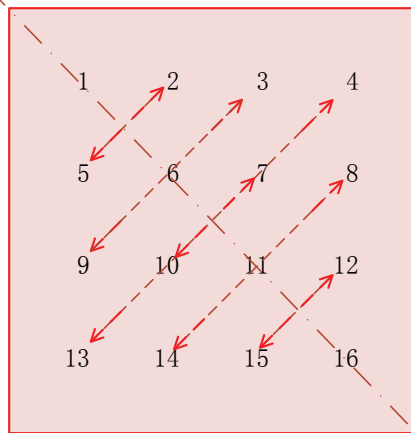


图 5-51 运行效果

对角线



对角线

图 5-52 行列转置

### 【关键代码】

```
cout<<"请输入一个 4*4 的二维数组: "<<endl;
for(i=0;i<row;i++)
{
    for(j=0;j<row;j++)
    {
```



```
        cin>>sz[i][j];
    }
}
cout<<"初始状态: "<<endl;
for(i=0;i<row;i++)
{
    for(j=0;j<row;j++)
    {
        cout<<sz[i][j]<<"    ";
    }
    cout<<endl;
}
for(i=0;i<row;i++)
{
    for(j=i+1;j<row;j++)
    {
        temp=sz[i][j];
        sz[i][j]=sz[j][i];
        sz[j][i]=temp;
    }
}
cout<<"翻转后的状态: "<<endl;
for(i=0;i<row;i++)
{
    for(j=0;j<row;j++)
    {
        cout<<sz[i][j]<<"    ";
    }
    cout<<endl;
}
```

【题目 5-9】理解下述程序代码，并补充空白。

读入一组数字，并进行累加计算，输出总和。程序代码如图 5-53 所示，程序的执行效果如图 5-54 所示。

```
#include <iostream>
using namespace std;
int main( )
{
    int a[1000];
    int n=0;
    while(____>>a[n])
    {
        n++;
    }
    int sum=____;
    for(int i=0;i<n;i++)
    {
        sum=____+a[i];
    }
    cout<<____<<endl;
    return 0;
}
```

图 5-53 求输入数字的累加和

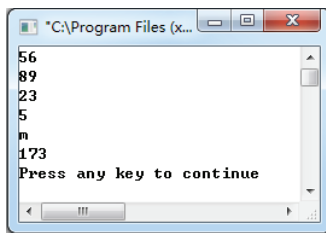


图 5-54 运行效果



【题目分析】图 5-43 所示的程序代码中定义了一个整型一维数组，但并没有对数组进行赋值。对数组进行赋值时要逐个进行，程序中用一个 `while` 循环给数组赋值。若给数组的某个元素进行赋值时不成功，则退出循环。`sum` 用来存放数组的累加和，故在使用前需要初始化为 0，若不初始化，则输出的结果是不可预料的。输出累加和，只要将 `sum` 的值输出即可。

【题目 5-10】设有数组定义 “`char array[]="Hello C++ Language";`”，编写程序计算数组 `array` 所占的空间。程序执行效果如图 5-55 所示。

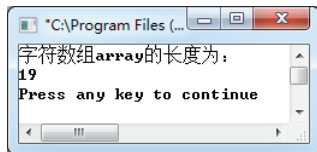


图 5-55 运行效果

【题目分析】该题目考查了字符串和字符数组的区别。要特别注意 C++ 中的字符串常量末尾会有一个结束符 `'\0'`。题目中用字符串常量初始化字符数组，则字符数组的长度应该是字符串的长度加 1，即加上串末尾的 `'\0'`。

#### 【关键代码】

```
char array[]="Hello C++ Language";
int k=1,i=0;
while(array[i]!='\0')
{
    i++;
    k++;
}
cout<<"字符数组 array 的长度为: "<<endl;
cout<<k<<endl;
```

# 第6章 函数

函数是 C++ 程序的主要组成部分。C++ 程序通过函数将变量、常量、表达式和语句等程序的基本要素结合在一起，完成一定的功能。本章将主要讲述函数的基本概念、函数的声明和定义，以及函数的参数传递、函数的调用和递归、main() 函数。

## 6.1 函数概述

本节简要介绍函数的基本概念、函数的分类等基本知识。

### 6.1.1 函数的基本概念

数学中的函数可以根据输入的数值求得一个确定的值。C++ 中的函数涵盖的意义更加广泛，如图 6-1 所示。

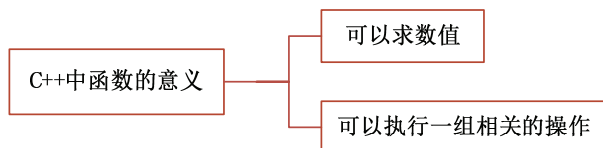


图 6-1 C++ 中函数的意义

简单地说，函数就是对复杂问题的一种“自顶向下，逐步求精”思想的体现。用户可以将一个大而复杂的程序分解为若干个相对独立而且功能单一的小块程序（函数）进行编写，并通过在各个函数之间进行调用来实现总体的功能，如图 6-2 所示。

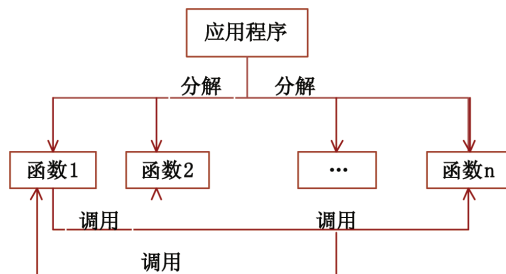


图 6-2 函数——程序的最小模块

事实上，设计一个 C++ 程序的过程就是编写函数的过程，至少也要编写一个 main() 函数。





执行 C++ 程序，也就是执行相应的 `main()` 函数。即从 `main()` 函数的第一个 “{” 开始，依次执行后面的语句，直到最后一个 “}” 为止。函数可以被一个函数调用，也可以调用另一个函数，它们之间可以存在着调用上的嵌套关系。



**注意：**函数是语句的集合，但不是任何语句的集合都是函数（语句块就不是函数）。

函数的本质有以下两点：

- (1) 函数由能完成特定任务的独立程序代码块组成，如有必要，也可调用其他函数。
- (2) 函数内部工作对程序的其余部分是不可见的。

### 6.1.2 函数的分类

在 C++ 程序设计中使用函数，能够使程序的层次结构清晰，便于程序的编写、阅读和调试。一般来说，在 C++ 中可从不同的角度对函数分类，如图 6-3 所示。

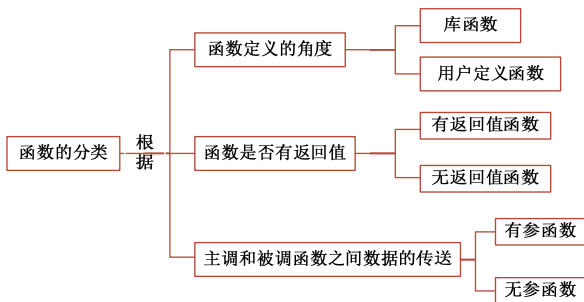


图 6-3 函数的分类



**注意：**在实际程序中，`#include` 命令后一般使用 `<>` 符号来调用库函数，使用 `"` 符号来调用自定义函数。



## 6.2 函数的组成

函数由返回类型、函数名、参数列表和函数体 4 部分构成，如图 6-4 所示。

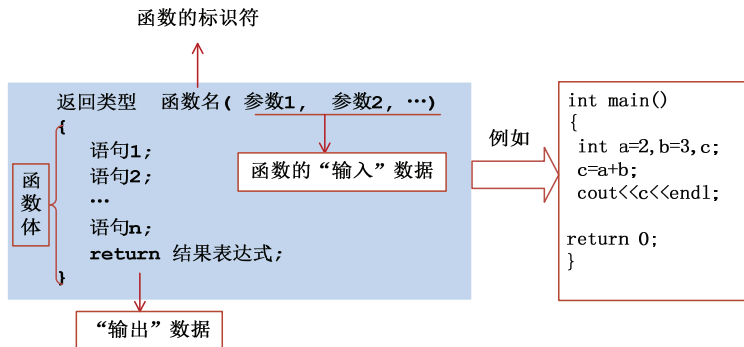


图 6-4 函数的组成



### 6.2.1 函数头

图 6-4 中的第一行“返回类型 函数名（参数列表）”称为函数头，定义了函数和调用它的函数之间的接口。

#### 1. 函数名

上级函数通过函数名实现对函数的调用，函数名是一个符合 C++语法要求的标识符。定义函数名与定义变量名的规则是一样的，但应尽量避免用下画线开头，因为编译器常常会定义一些以下画线开头的变量和函数。函数名应尽可能反映函数的功能。

#### 2. 参数列表

0 个或多个变量，用于向函数传递数值或从函数带回数值，每个参数都应采取“类型 变量名”形式，参数列表中的参数称为形式参数，简称形参。关于形参和实参稍后会详细介绍。

#### 3. 返回类型

指定函数由 `return` 返回的函数值的类型，如果函数没有返回值，返回类型应为 `void`。C++ 对返回值的类型有一定的限制，它不能是数组，但可以是其他任何类型，如整型、浮点型及指针，甚至是结构和共用体等。



**注意：**C++不允许返回数组，但却可以把数组当做结构体的组成部分返回。

### 6.2.2 函数体

花括号中的语句称为函数体，一个函数的功能，通过函数体中的语句来完成，函数体指明了要进行的操作和操作顺序。程序执行到函数体中的 `return` 语句返回，返回语句的基本形式如图 6-5 所示。

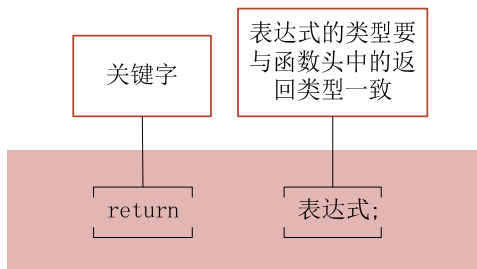


图 6-5 `return` 语句

在函数体中可以有多条 `return` 语句，但函数只能有一个出口，也即只执行一条返回语句。如图 6-6 的所示程序代码中，`max` 函数中有两条返回语句。但是，函数执行时返回语句只会执行一条，也就是说一旦执行了返回语句，返回语句后的代码就不会被执行，函数也就运行完毕了。

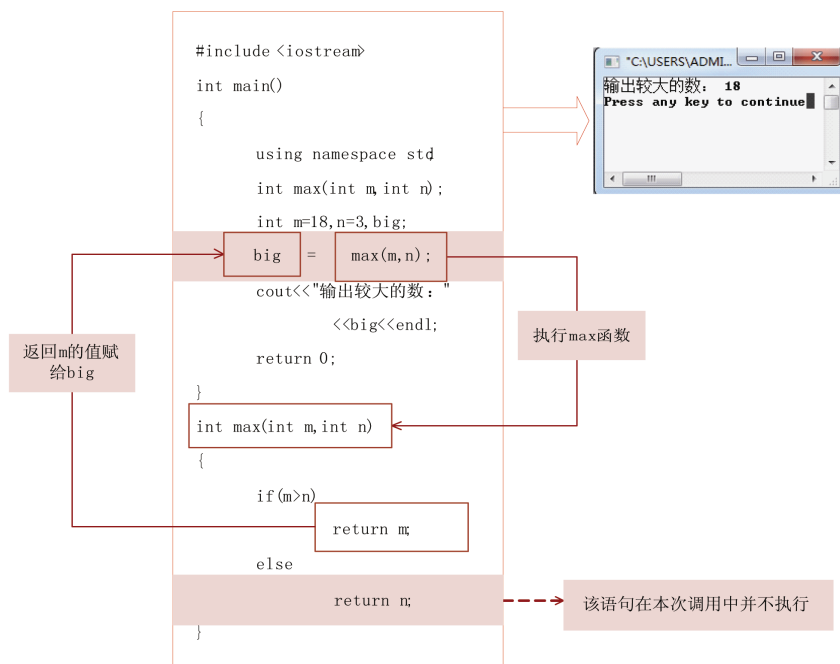


图 6-6 函数的返回语句

为了将图 6-6 中函数 max 的执行情况理解得更清楚，下面给出函数 max 的流程图，如图 6-7 所示。

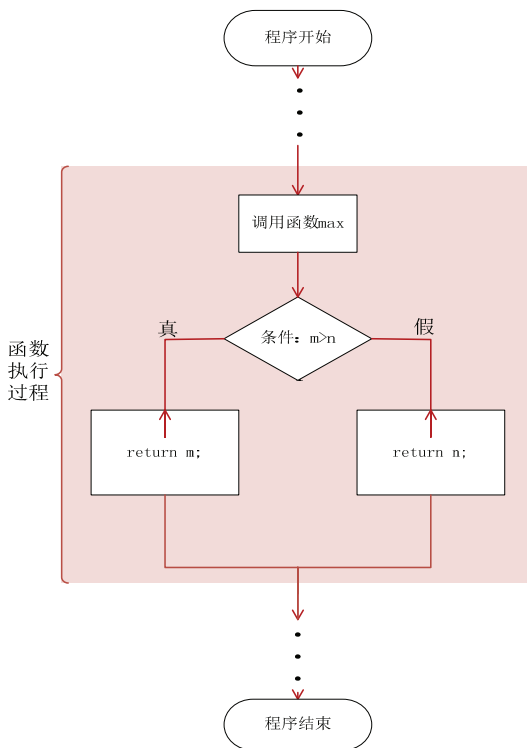


图 6-7 函数执行的流程图



return 语句中若表达式与返回类型不一致，则编译器会根据函数头中指定的返回类型对表达式进行转换。返回语句主要起以下 3 个作用：

- (1) 撤销函数调用时为参数和变量分配栈内空间。
- (2) 向调用函数（上级）返回最多一个值（表达式的值）。
- (3) 将程序流程从当前函数返回上级函数。



## 6.3 函数的声明和定义

在 C++ 程序中调用函数之前，首先要对函数进行声明。函数声明就是为了告诉编译器存在这样一个函数，并且可以在后面的程序中使用。函数定义是在函数声明的基础上多加函数体。本节将详细讲解函数的声明和定义。

### 6.3.1 函数原型——函数的声明

函数的声明又称函数原型。函数声明的作用及声明函数的一般形式如图 6-8 所示。

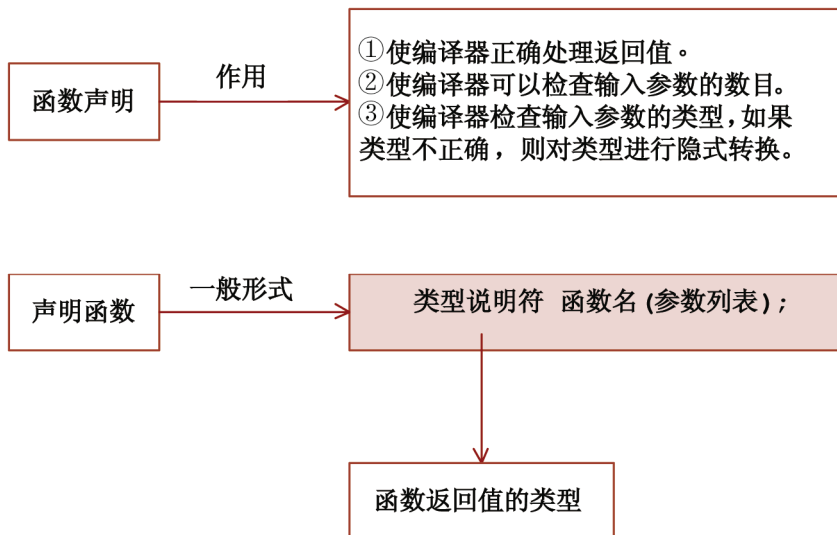


图 6-8 函数声明的作用及声明函数的一般形式

需要注意的是，在函数的声明和定义中，不能没有参数列表。如果函数不需要参数，则可以用空参

数列表或只带一个 void 关键字的参数列表，例如，图 6-9 中两种表示形式是等价的。

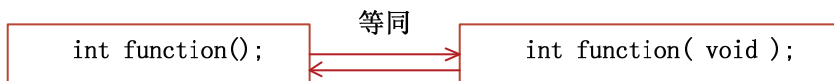


图 6-9 空参数列表的表示形式

函数声明时，参数列表中的每个参数可以有名称，也可以没名称，即只带一个类型关键字即可。例如，图 6-10 中两种表示形式是等价的。

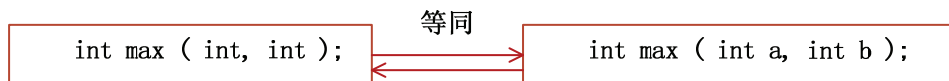


图 6-10 参数名称的表示形式



**注意：**函数声明也是一条语句，后面必须带有“;”标识语句的结束，否则编译器将给出错误信息提示。

### 6.3.2 函数实现——函数的定义

函数的定义又称函数实现。一个函数的定义由返回类型、函数名、参数列表和函数体组成。根据函数是否需进行参数传递，函数的定义又分为有参函数和无参函数的定义，如图 6-11 所示。

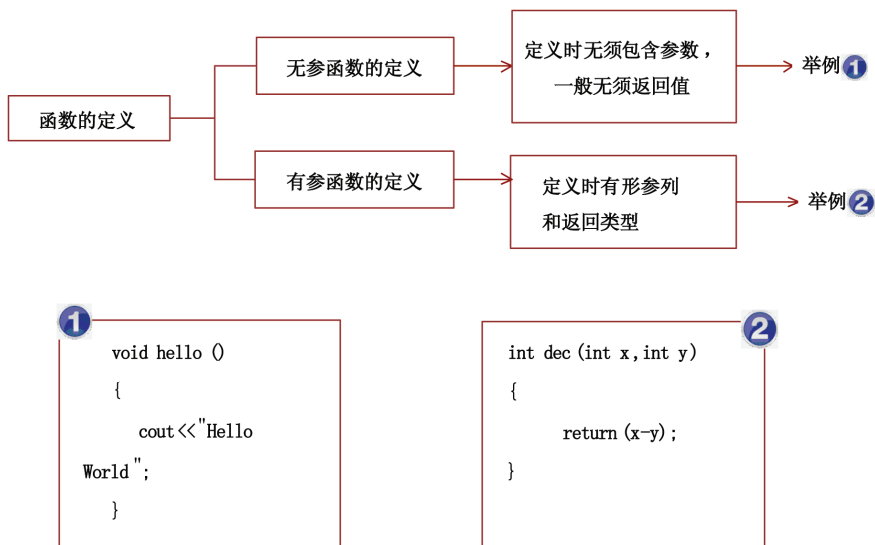
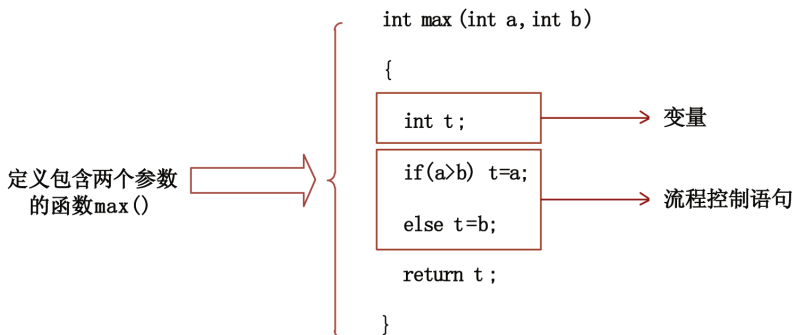


图 6-11 函数的定义

此外，在函数定义的函数体中可以定义其他变量，或使用前面章节讲解到的流程控制语句，用以完成函数的功能。例如，图 6-12 所示的程序中定义的函数 `max()`，返回两个数之间的较大数。

图 6-12 函数 `max()`



如果在调用前函数已经定义，则不必另外声明。下面通过图 6-13 中的代码体验一下。

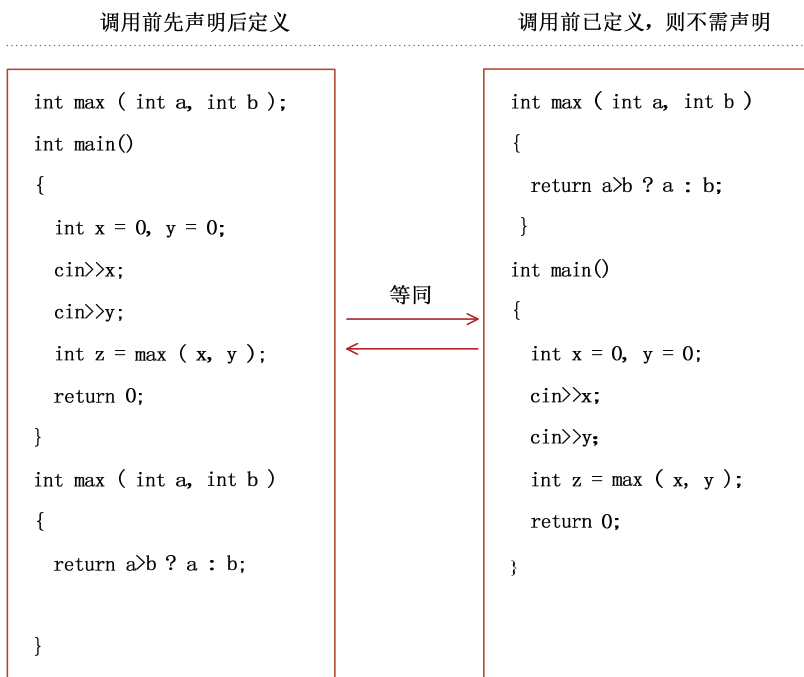


图 6-13 定义在调用前后的区别

若被调用函数的定义在调函数定义后面，则在调用函数中需要对被调函数进行声明。否则，不需要声明。如图 6-13 左边的代码段，max()函数定义在使用之后，所以对 max()函数事先进行了声明；而对右边的代码，max()函数定义在前，使用在后，所以声明可以省略。



## 6.4 函数参数传递

在函数的调用中是通过参数的传递实现函数具体的功能的。本节将对 C++的函数参数及值传递做具体介绍。

### 6.4.1 函数的形参和实参

一般来说，C++中函数的参数分为形式参数（形参）和实际参数（实参）两种，如图 6-14 所示。

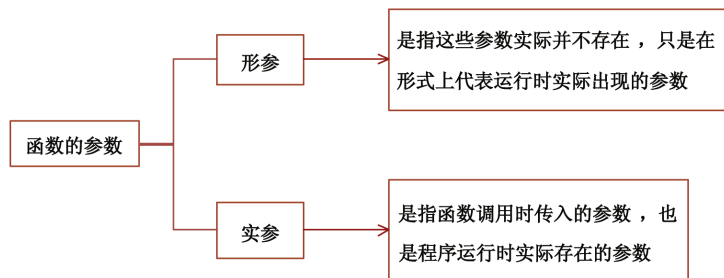


图 6-14 C++中函数的参数



简单地说，被调用函数与主调用函数之间的通信可以通过参数的传递来实现。在具体程序中，形参和实参的分辨非常容易，如图 6-15 所示的程序代码中就包含了形参和实参。

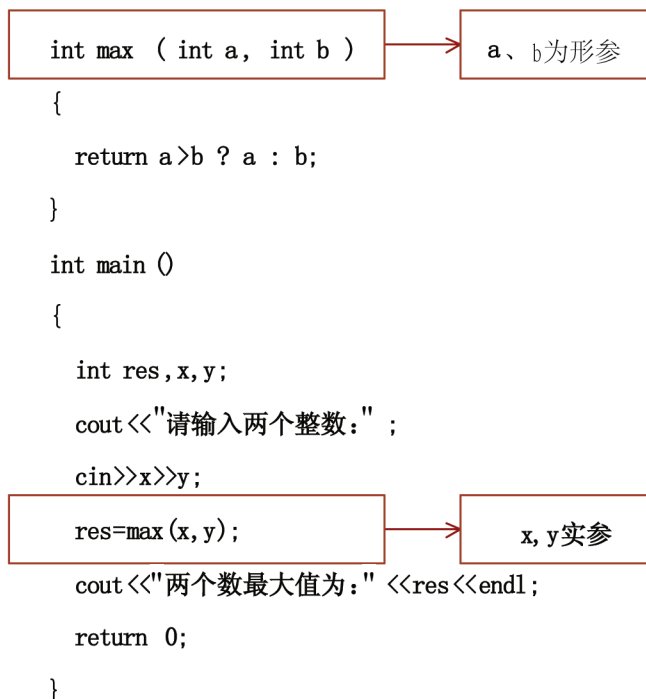


图 6-15 形参与实参

在参数传递中用户必须遵循如图 6-16 所示的规则。

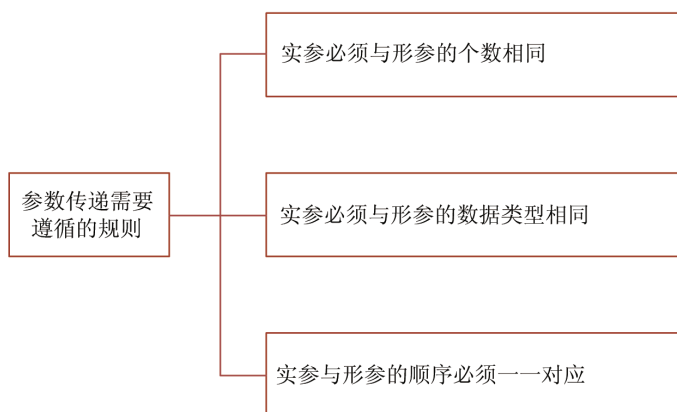


图 6-16 参数传递需遵循的规则

## 6.4.2 值传递

值传递是指实参传递给函数后，系统构建一份实参的副本，其值与实参的值相同。此后函数将针对这份副本进行操作，对原始的实参没有任何影响。



值传递过程中，被调函数的形参作为被调函数的局部变量处理，即在堆栈中开辟了内存空间以存放由主调函数放进来的实参的值，从而形成了实参的一个副本。值传递的特点如图 6-17 所示。

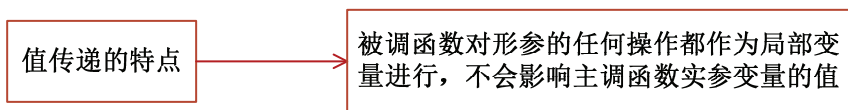


图 6-17 值传递的特点

【示例 6-1】下面的程序定义了一个交换两个数的函数 swap()，实参与形参传递时采用的是值传递方式，实现代码及结果如图 6-18 所示。

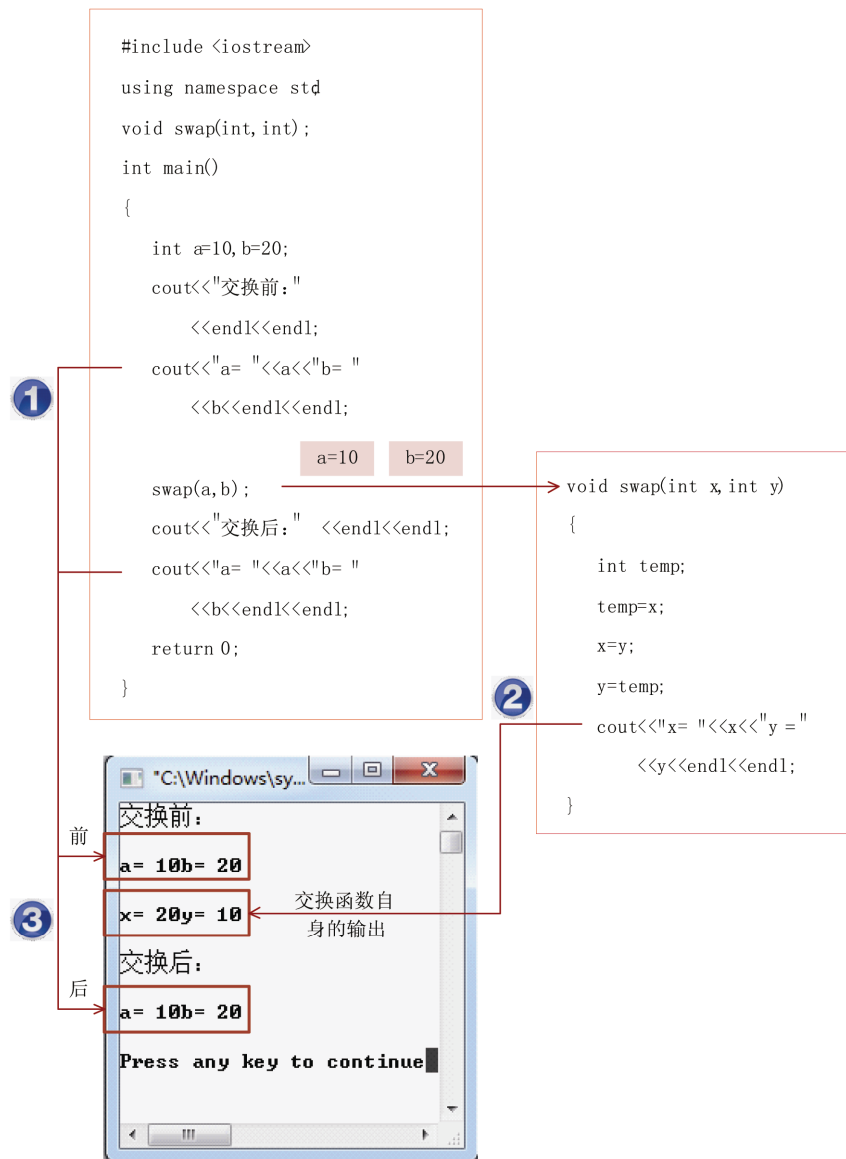


图 6-18 值传递实例





可以看出,上述代码预期完成两个变量 **a** 和 **b** 的交换,但图中返回的结果却没有实现交换,这是因为实参和形参的传递方式为值传递,传递的只是一个副本,所以修改不对实参产生效果。调用函数和被调函数发生的唯一数据关联就是实参向形参赋值,此后,被调用函数中的变量(含形参)与调用函数中的变量便各有各的内存空间,互不干涉,即使变量名相同,也会被编译器认为是不同的变量。

换句话说,被调函数中执行的任何操作都只作用在被调函数内的变量上,而不作用在调用函数的变量上,通过传值调用时被调函数无法改变调用函数中的变量值。



## 6.5 函数的调用

进行函数的声明或定义后,在其他程序中就可以对该函数进行调用了。本节将详细介绍函数的调用过程、无参和带参函数的调用、默认形参值的调用及嵌套调用,以及特殊的数组参数的调用。

### 6.5.1 函数的调用过程

一般来说,C++程序都是从主函数 `main()` 开始执行,当执行到函数调用语句时,就会转去执行调用函数,执行后仍然返回到主函数,直至程序结束。当调用一个函数时,整个调用过程分为3步进行,如图 6-19 所示。

调用函数过程3步走



图 6-19 函数调用过程 3 步走

例如,如图 6-20 所示为在函数 `func1()` 中调用函数 `func2()` 的过程。

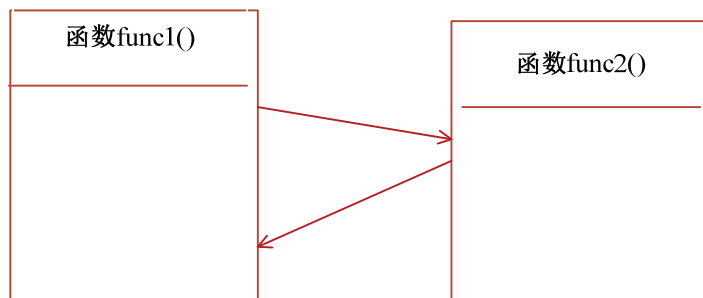


图 6-20 在函数 `func1()` 中调用函数 `func2()` 的过程

### 6.5.2 无参函数的调用

当函数的参数列表为空时,该函数为无参函数。在 C++ 中,调用无参函数不涉及形参和实参,调用方式非常简单。一般来说,C++ 中无参函数的调用形式如下:

```
<函数名> ();
```



此外，无参函数一般也没有返回值，因此在函数中调用一个无参函数的功能如图 6-21 所示。



图 6-21 调用无参函数的功能

【示例 6-2】下面的程序定义了两个无参函数 `hello()` 和 `welcome()`，并在主函数 `main()` 中调用这两个函数，实现代码及结果如图 6-22 所示。

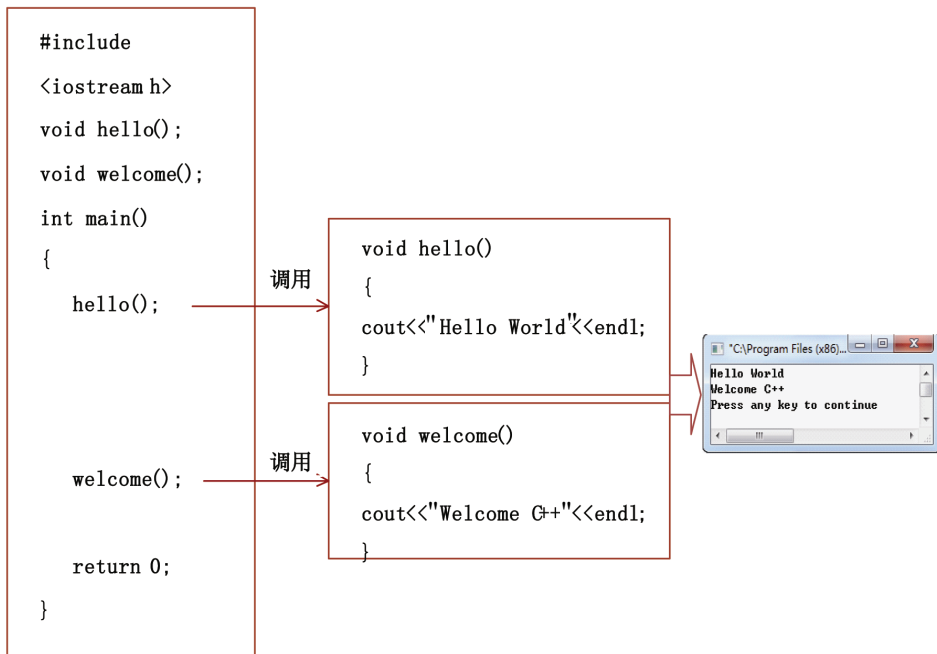


图 6-22 无参函数调用实例



**注意：**调用无参函数时，函数名后的 `()` 不能省略。

### 6.5.3 带参函数的调用

在具体的程序设计中，函数一般都有参数列表和返回值，在调用带参函数时，需要注意其参数的类型、数目和顺序必须是一一对应的。

【示例 6-3】下面示例定义一个函数判断用户输入的年份是否为闰年，并在主函数 `main()` 中调用该函数，实现代码及结果如图 6-23 所示。

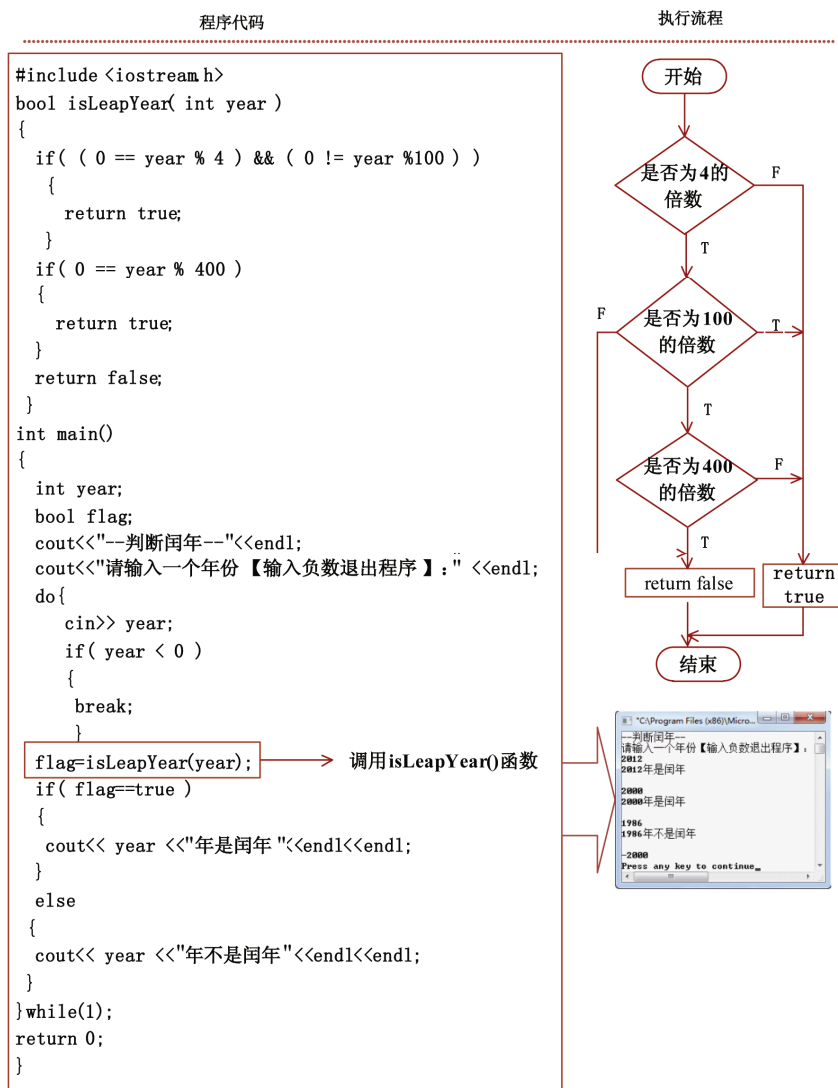


图 6-23 闰年的判断

### 6.5.4 默认形参值的调用

在 C++ 中，调用函数时通常要为函数的每个形参给定对应的实参。若没有给出实参，则按指定的默认值进行工作。当一个函数既有定义又有声明时，形参的默认值必须在声明中指定，而不能在定义中指定。只有当函数没有声明时，才可以在函数定义中指定形参的默认值。

此外，默认值的定义必须遵守从右到左的顺序，如果某个形参没有默认值，则它左边的参数就不能有默认值，如图 6-24 所示。

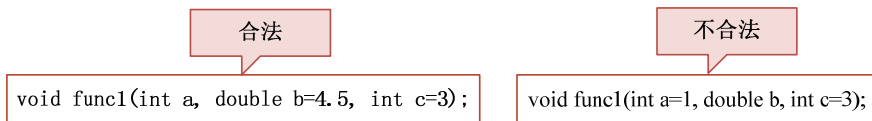


图 6-24 默认值的定义遵守规则——从右到左



在进行函数调用时，实参与形参进行匹配的顺序如图 6-25 所示。



图 6-25 实参与形参进行匹配的顺序

如图 6-26 所示的代码定义了 ShowString() 函数用于字符串的输出，该函数有 3 个参数，分别是字符串 ptext（采用指针传递）、输出次数 n（int 型，值传递）、每行缩进字数 m（int 型，值传递）。ShowString() 函数用于将字符串 ptext 输出 n 遍，每遍首字母依次缩进 m 个空格。

```
#include <iostream>
#include <iomanip>
#include <string>
using namespace std;
int main()
{
    void ShowString(char*, int n=2, int m=5);
    ShowString("I Love C++");//默认两个参数调用
    cout<<"-----"<<endl;
    ShowString("Hello", 1);//默认最后一个参数
    cout<<"-----"<<endl;
    ShowString("World", 3, 2);//全部参数都设置
    return 0;
}

void ShowString(char* ptext, int n, int m)
//函数定义，输出n遍字符串，每遍缩进m个字符
{
    int len=strlen(ptext);
    for(int i=0; i<n; i++)
    {
        cout<<setw(len)<<ptext<<endl;
        len=len+m;
    }
}
```

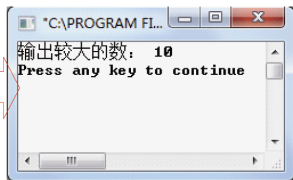


图 6-26 默认参数实例

关于默认参数调用要遵守下面 4 条规则。

(1) 参数默认必须按从后向前的顺序。即在函数声明时，如果对第 n 个参数进行初始化，那么其后面所有的参数都应该进行初始化。所以，在函数定义时，应该合理安排形参列表顺序。



(2) 和函数声明一样，在函数调用时省略某个参数，必须省略其后所有后续参数，下列函数调用是不合法的：

```
ShowString("world",,2);
```

(3) 除非必要，否则不要在函数内（尤其是函数的第一行）对参数进行初始化，否则，函数调用时实参向形参的传递就没有意义。例如，图 6-27 所示的代码中，实参的值无法传递给形参。

```
#include <iostream>
#include <iomanip>
#include <string>
using namespace std;
int main()
{
    int max(int a,int b);      "
    cout<<("输出27和16中的较大数:")
;
    int big=max(27,16);
    cout<<big<<endl;
    return 0;
}
int max(int a,int b)
{
    a=12;
    b=7;
    if(a>b)
        return a;
    else
        return b;
}
```

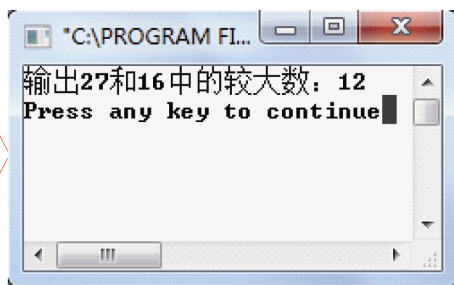


图 6-27 参数的传递

图 6-27 中的代码原意是要输出 27 和 16 两者中的较大数，结果却输出了 12，实际上，不论实参的值是多少，函数的返回值都会是 12。



(4) 在函数调用时，也可以对实参变量初始化，但此时不涉及默认参数调用的问题。实际上，每个实参都可以看做一个表达式，首先计算参数表达式的值，并将此值传递给形参。例如，“ShowString("world",int i=3,2);”等价于“ShowString("world",3,2);”。

## 6.5.5 嵌套调用

嵌套调用是一种特殊的函数调用方式，即在一个函数的函数体中又调用了另外一个函数。例如，下面定义了一个函数 func1()，而在 func1()的函数体中调用了 func2()，这就是函数的嵌套调用，代码及调用流程如图 6-28 所示。

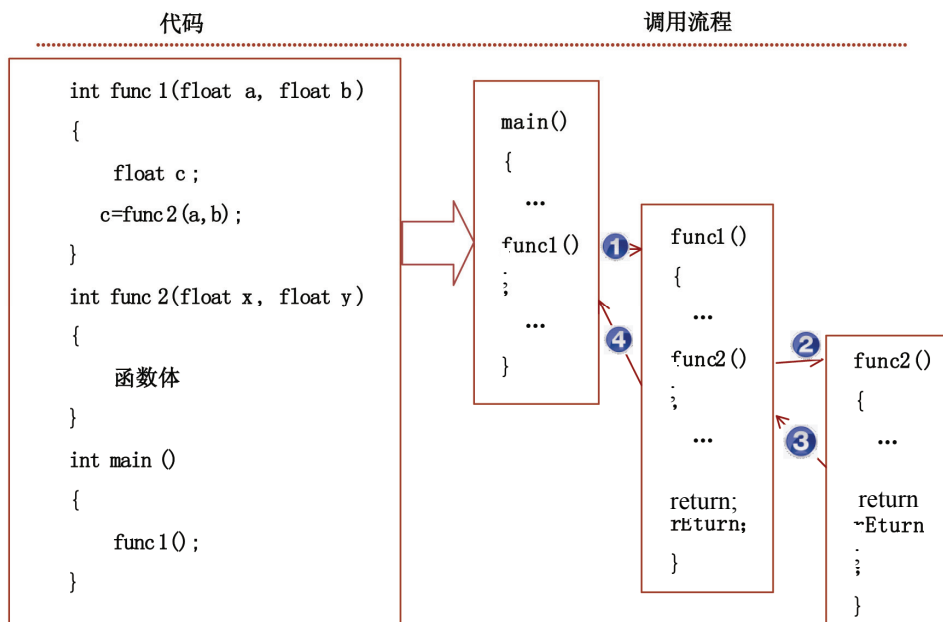


图 6-28 函数嵌套



**注意：**函数 func1()和 func2()是分别独立定义的函数，互不从属。并且在 func1()中调用 func2()函数前，func2()函数已经被声明。

【示例 6-5】下面的程序计算表达式  $s=(12)!+(22)!+(32)!$ ，即计算 1~3 的平方的阶乘之和。该示例的实现需要定义两个函数，一个用于计算平方值，另一个用于计算阶乘值，其实现代码及结果如图 6-29 所示。



图 6-29 计算 1~3 的平方的阶乘之和

### 6.5.6 数组作为函数参数

前面的示例中介绍了变量可以作为函数的实参，本小节主要讲解数组元素和数组名作函数实参。

#### 1. 数组元素作函数实参

数组元素作为函数的实参和用变量作实参一样，是一种单向传递，即由实参传向形参——值传递方式。

【示例 6-6】该示例对已经初始化的一维整型数组 `a[5]` 的每个元素乘以 2，输出乘以 2 后每个元素的值。该程序采用一维数组元素作为函数实参，其实现代码及结果如图 6-30 所示。

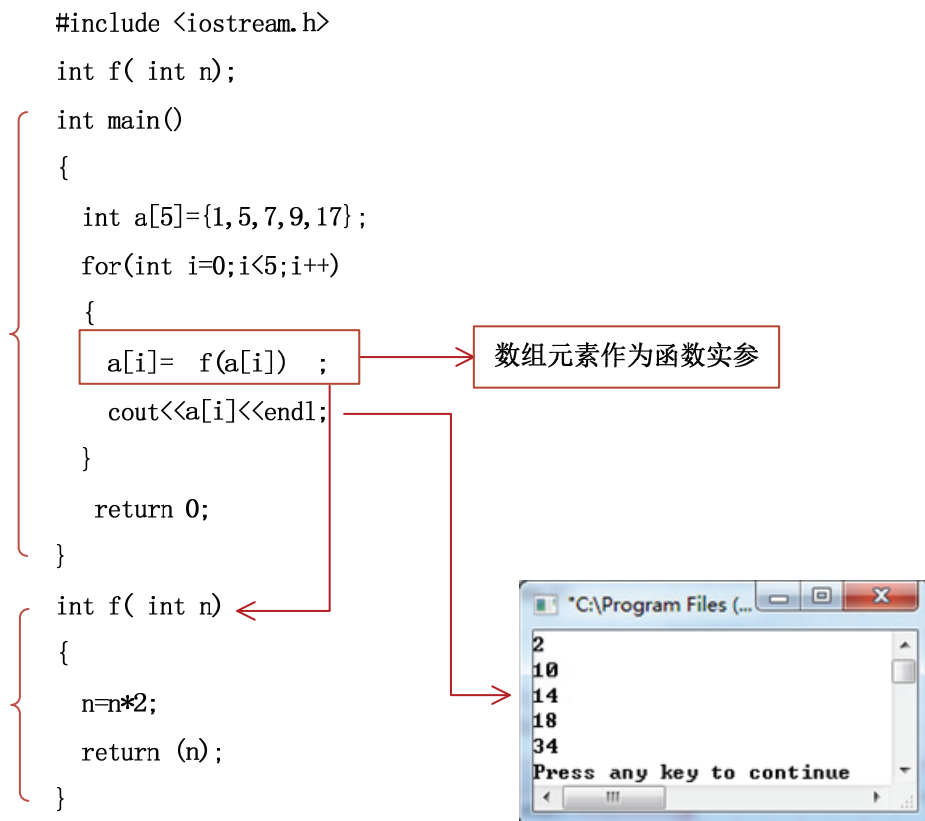


图 6-30 数组元素作函数实参

## 2. 数组名作函数实参

除了数组元素可以作为函数的实参，数组名也可以作为函数的实参。数组名作为函数实参与数组元素作为函数实参的区别是“地址传递参数形式”。

【示例 6-7】该示例对【示例 6-6】作了改变，使用数组名作函数实参。程序代码及输出结果如图 6-31 所示。



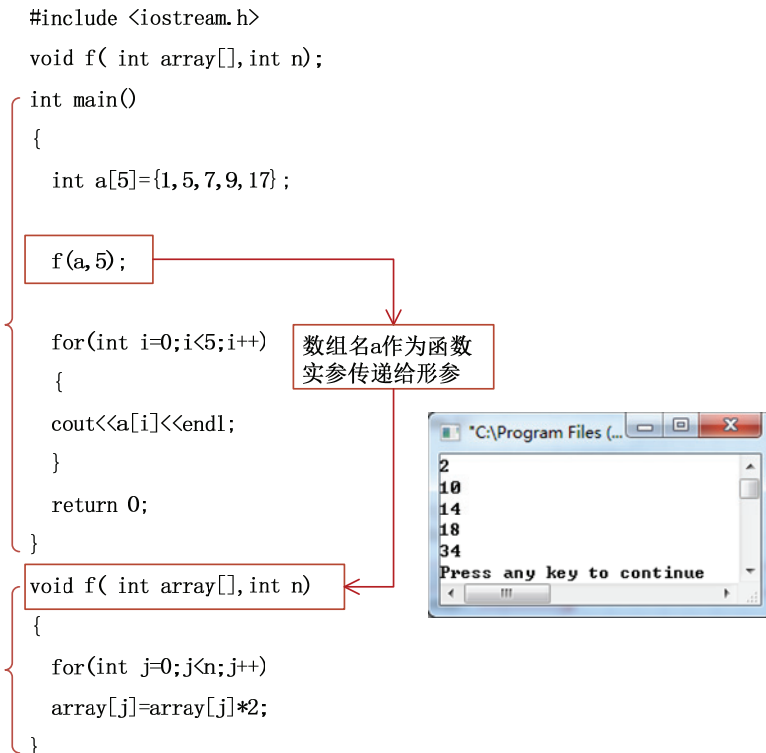


图 6-31 数组名作函数实参



## 6.6 递归函数

函数直接调用自己或通过一系列调用语句间接调用自己，称为函数的递归调用，常用来解决结构自相似的问题。结构自相似的概念如图 6-32 所示。

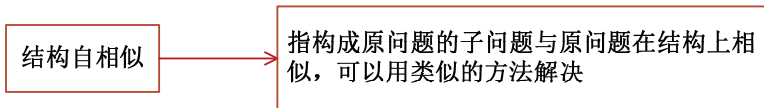


图 6-32 结构自相似的概念

实际上，递归是把一个不能或不好解决的大问题转化为一个或几个小问题，再把这些小问题进一步分解成更小的问题，直至每个小问题都可以直接解决。因此，递归有两个基本要素，如图 6-33 所示。

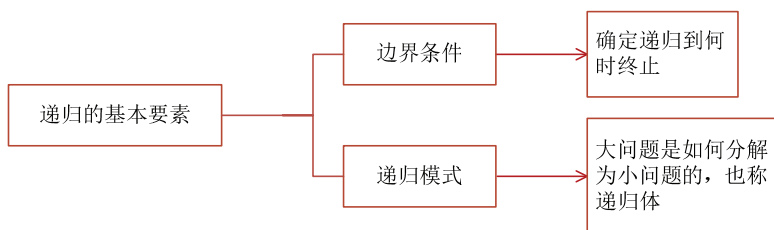


图 6-33 递归的两个基本要素



递归函数只有具备了这两个要素，才能在有限次计算后得出结果。C++语言的特点之一就在于允许函数的递归调用。递归分为直接递归和间接递归，下面将具体介绍。

## 6.6.1 直接递归

在递归调用中，直接递归是指函数直接调用自身。对于函数 `func()` 而言，直接递归的调用过程如图 6-34 所示。

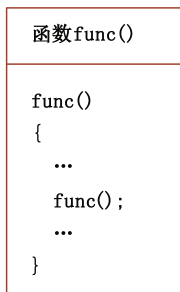


图 6-34 直接递归的调用过程

【示例 6-8】下面程序接收用户从键盘输入的一个整数 `n`，编写一个函数求该整数 `n` 的阶乘，其采用的就是函数的直接递归，其实现代码及结果如图 6-35 所示。

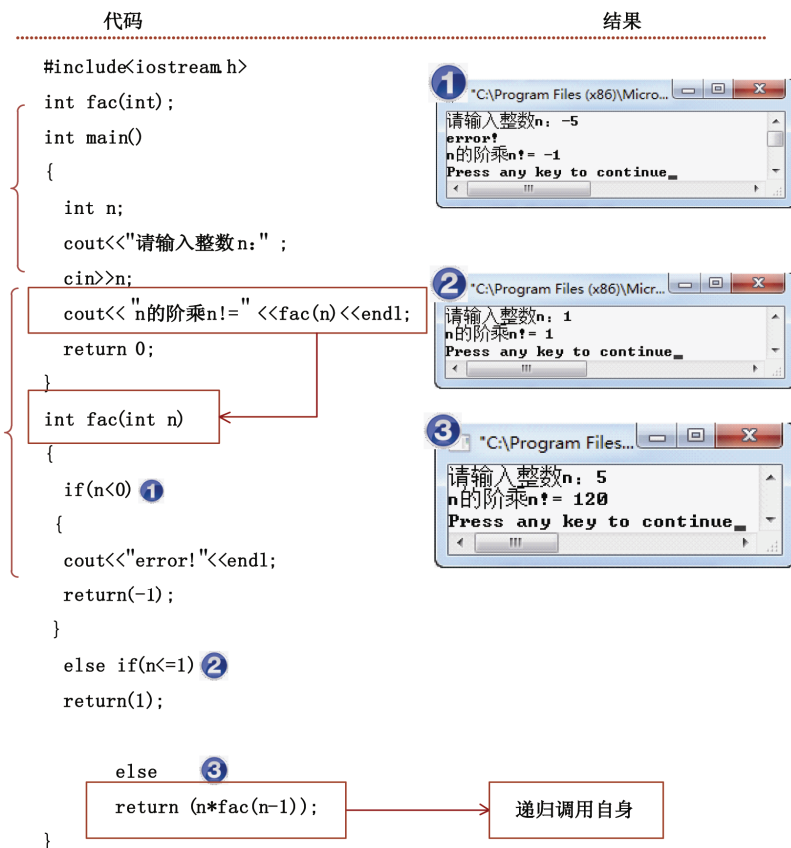


图 6-35 直接递归实例



## 6.6.2 间接递归

间接递归也是递归的一种重要形式，它是指 A 函数调用了 B 函数，而 B 函数又调用 A 函数，如图 6-36 所示。

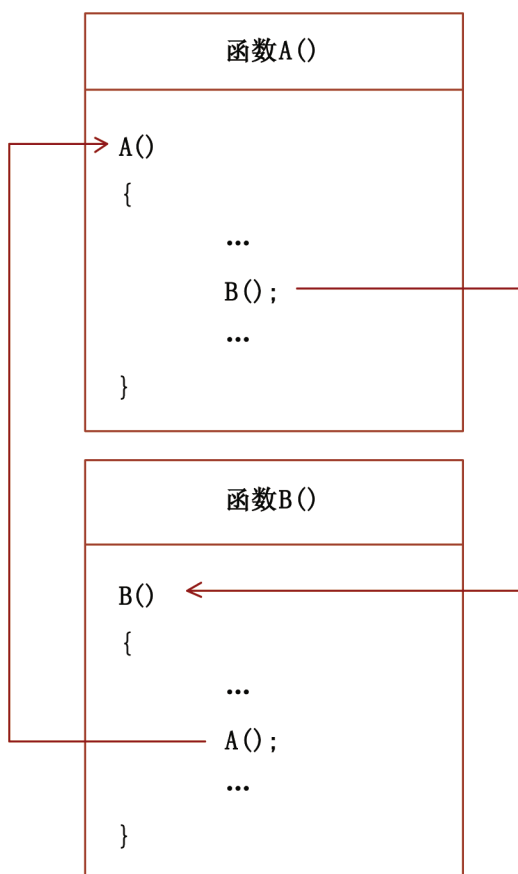


图 6-36 间接递归形式

一般来说，由于间接递归不停地在两个函数中相互调用，容易造成死循环，必须加上结束循环的条件。间接递归在实际程序中应用不多，一般在一些特殊情况下使用。

【示例 6-9】下面程序使用到函数 f()和 g()之间的间接递归，实现代码及结果如图 6-37 所示。



```
#include<iostream h>
```

```
int f(int n);
```

```
int g(int n);
```

```
int main()
```

```
{
```

```
    int i;
```

```
    for(i=1;i<6;i++)
```

```
        cout<<f(i)<<"", "<<g(i)<<endl;
```

```
    return 0;
```

```
}
```

```
int f(int n)
```

```
{
```

```
    if(n==1)
```

```
        return 1;
```

```
    else
```

```
        return n*g(n-1);
```

```
}
```

```
int g(int n)
```

```
{
```

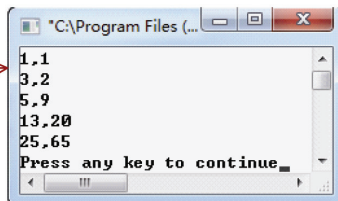
```
    if(n==1)
```

```
        return 1;
```

```
    else
```

```
        return n*f(n-1);
```

```
}
```



递归调用g()

递归调用f()

图 6-37 间接递归实例

需要注意的是，递归的使用能够大大简化程序的结构，便于读者理解。使用函数的递归调用需要注意 3 个方面，如图 6-38 所示。

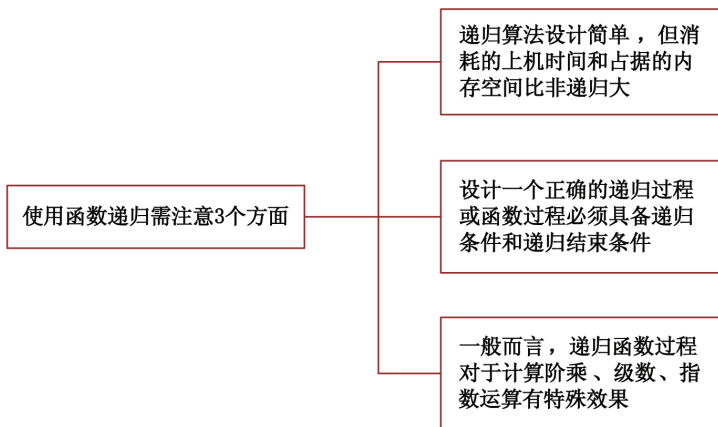


图 6-38 使用函数递归调用需注意 3 个方面



## 6.7 main()函数

每个 C++ 程序都必须有一个 `main()` 函数，`main()` 函数又称为主函数，是 C++ 程序中最重要函数，所有完整可运行的 C++ 程序都必须有一个唯一的 `main()` 函数。

### 6.7.1 不带参数的 `main()` 函数

在前面章节的示例中，用到了 `main()` 函数，其都没有带任何参数和返回值。如图 6-39 所示为 `main()` 函数的最简单形式。

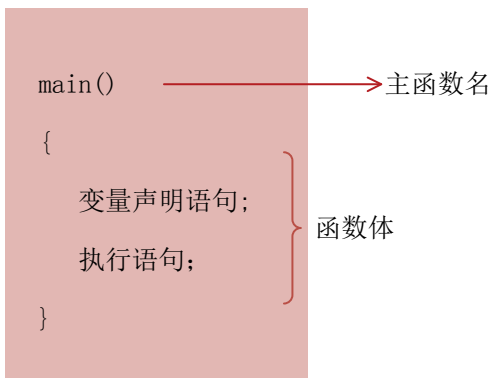


图 6-39 `main()` 函数的最简单形式

在具体使用中，对于一些简单的问题，只用一个 `main()` 函数即可，程序的全部处理语句都放在其中。而对于一些复杂的问题，需要进行模块化设计，即把一个复杂的问题分解成若干个相对简单的子问题。每个子问题由一个或多个函数来处理，而 `main()` 函数负责总控，调用相应的函数。

### 6.7.2 带参数的 `main()` 函数

事实上，`main()` 函数是一个特殊的函数，其中“`main`”是函数名，与其他函数一样，该函数也可以有返回值和参数表。在 `main()` 函数中允许带两个参数，一个是整型数据类型 `argc`，另一个是指向字符型的指针数组 `argv[]`。这两个参数在 `main()` 函数头部声明的格式如图 6-40 所示。

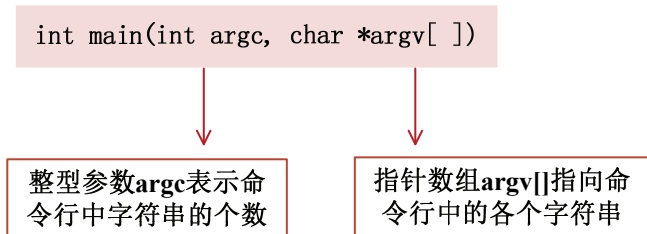


图 6-40 带参数的 `main()` 函数的格式



**说明：**这两个参数可以用任何合法的标识符命名，但习惯上采用 `argc` 和 `argv` 表示。



## 6.8 函数的综合应用

在许多数学运算中,表达式都有一定的规律性,这种特征的计算就可以用递归函数来实现。

【示例 6-10】下面程序用递归法求:  $(x/1!)+(x^3/3!)+(x^5/5!)+\dots+(x^{2n-1}/(2n-1)!)$ , 到第  $n$  项,  $n$  和  $x$  的值由键盘输入。其实现代码及结果如图 6-41 所示。

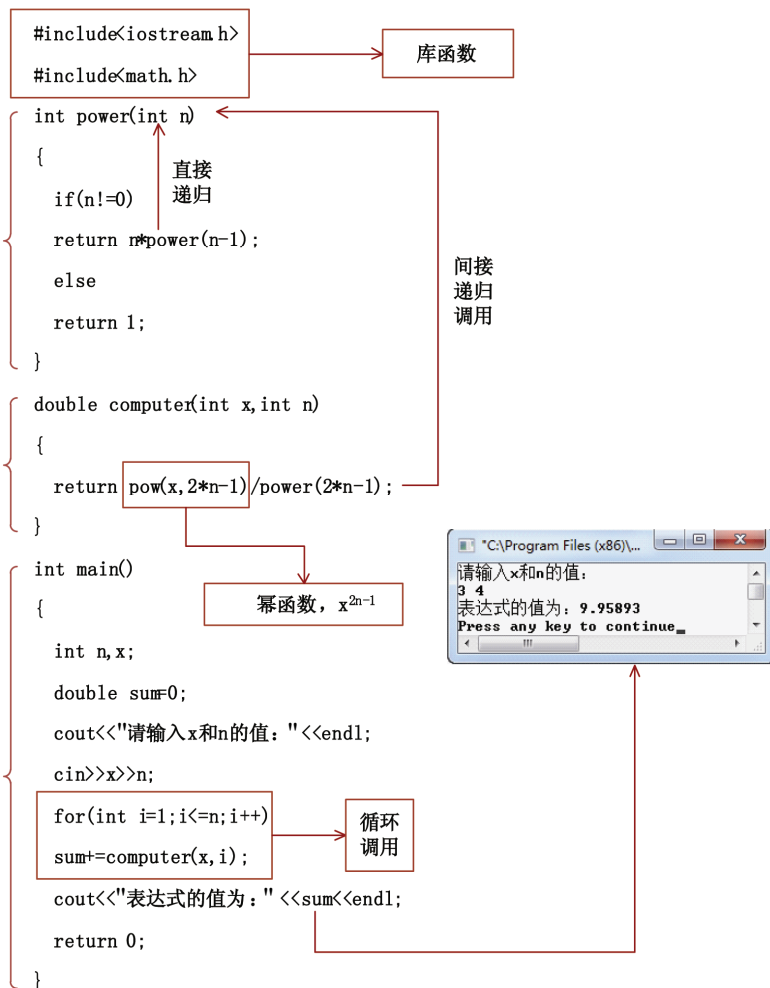


图 6-41  $(x/1!)+(x \times x \times x/3!)+(x^5/5!)+\dots+(x^{2n-1}/(2n-1)!)$ 求法

## 6.9 小结

本章详细讲解了函数的定义、声明和调用,重点讲述了函数的参数传递、嵌套调用和递归调用等。简单介绍了 `main()` 函数带参数的形式,最后通过两个实例进一步掌握函数的应用。

## 6.10 习题

【题目 6-1】编写一个返回两个数中大者的函数 `max()`。分别将该函数定义放在调用函数之



前和调用函数之后，体会两种定义方式的不同。程序执行效果如图 6-42 所示。

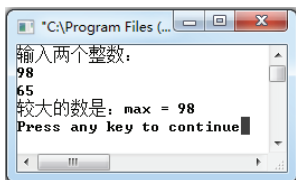


图 6-42 运行结果

**【题目分析】** 本题考查函数定义和函数的声明的基本知识。函数在使用之前必须先声明，才能使用。若函数的定义放在调用函数的前面，则在调用函数中不需要进行声明。否则，在调用函数中必须对被调函数进行声明。

**【关键代码】**

**max()函数定义在调用函数之前:**

```
int max(int a,int b)
{
    return a>b?a:b;
}
int main()
{
    int x=0;
    int y=0;
    cout<<"输入两个整数: "<<endl;
    cin>>x;
    cin>>y;
    int z=max(x,y);
    cout<<"较大的数是: max = "<<z<<endl;
    return 0;
}
```

**max()函数定义在调用函数之后:**

```
int main()
{
    int max(int ,int );
    int x=0;
    int y=0;
    cout<<"输入两个整数: "<<endl;
    cin>>x;
    cin>>y;
    int z=max(x,y);
    cout<<"较大的数是: max = "<<z<<endl;
    return 0;
}
int max(int a,int b)
{
    return a>b?a:b;
}
```

**【题目 6-2】** 按要求定义 3 个函数，并在主函数中调用它们。

函数一：定义一个无参数，无返回值类型的函数。函数可以输出两行字符串，它们分别是 "Welcome to Learn C++ Function!" 和 "My name is Wang."。

函数二：定义一个无参数，有返回类型的函数。函数根据输入的数据是否大于 0，返回真假。



函数三：定义一个有参数，也有返回值类型的函数。函数可以计算任意两个整数的乘积，并将乘积作为返回值返回。

运行效果如图 6-43 所示。

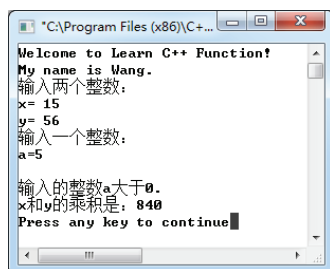


图 6-43 运行效果

【题目分析】 本题考查函数定义与函数调用的有关知识。

【关键代码】

```
int main()
{
    void Welcome();
    bool GetInt();
    int MultTwo(int,int);
    int x=0;
    int y=0;
    Welcome();
    cout<<"输入两个整数: " <<endl;
    cout<<"x= ";
    cin>>x;
    cout<<"y= ";
    cin>>y;
    bool b;
    b=GetInt();
    if(b)
        cout<<endl<<"输入的整数 a 大于 0." <<endl;
    else
        cout<<endl<<"输入的整数 a 小于或等于 0." <<endl;
    int z;
    z=MultTwo(x,y);
    cout<<"x 和 y 的乘积是: " <<z<<endl;
    return 0;
}

void Welcome()
{
    cout<<"Welcome to Learn C++ Function!" <<endl;
    cout<<"My name is Wang." <<endl;
}

bool GetInt()
{
    int a;
    cout<<"输入一个整数: " <<endl;
    cout<<"a=";
    cin>>a;
    if(a>0)
    {
        return true;
    }
    else
    {

```





```

        return false;
    }
}
int MultTwo(int x,int y)
{
    return x*y;
}

```

【题目 6-3】以下程序代码希望实现交换变量  $x$  和  $y$  的值，但是输出的结果却如图 6-44 所示。 $x$  和  $y$  并没有交换，请读者思考这么输出的原因。

```

#include <iostream>
using namespace std;
int main()
{
    void swap(int,int);
    int x=0;
    int y=0;
    cout<<"输入两个整数: "<<endl;
    cout<<"x= ";
    cin>>x;
    cout<<"y= ";
    cin>>y;
    swap(x,y);
    cout<<"交换后: "<<endl;
    cout<<"x= "<<x<<endl;
    cout<<"y= "<<y<<endl;
    return 0;
}
void swap(int a,int b)
{
    int temp;
    temp=a;
    a=b;
    b=temp;
}

```

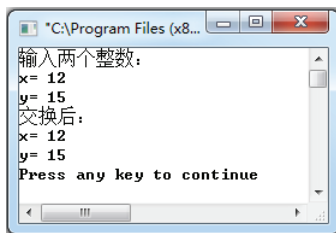


图 6-44 输出结果

【题目分析】这里主要考查函数参数的传递方式。实参在传递给函数时，是将实参的副本传递给形参，函数操作的变量并不是实参本身。因此，上述程序中的变量  $x$  和  $y$  并没有交换。也就是说实参与形参之间的传递是单向的。

#### 【关键代码】

```

    swap(x,y);
void swap(int a,int b)
{
    int temp;
    temp=a;
    a=b;
    b=temp;
}

```



上述代码只是将形参 `a` 和 `b` 进行了交换,但是在函数调用结束后,形参变量的内存空间就已经被释放了。这样看来,函数 `swap()` 实际上什么也没有做。

【题目 6-4】定义一个整型数组,编写一个函数,实现对这个整型数组按照递增的顺序进行排序。排序算法采用选择排序。要求:适当输出一些让用户输入数据的提示信息,另外,在主函数中将排好序的数组元素输出。程序的运行结果如图 6-45 所示。

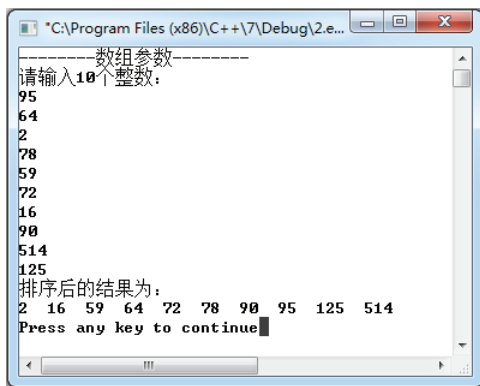


图 6-45 运行结果

【题目分析】本题主要考查函数的编写能力,本题要求数组作为函数的形参。在用数组作函数的形参时,要注意在函数的参数列表中添加一个参数,用以传递函数的长度信息。因为用数组作形参,函数并不能确定数组的长度,即使在形参数组中注明长度。另外,本题还考查了选择排序算法。选择排序算法的基本思想是:依次确定第  $i$  ( $i=0, 1, \dots, n-1$ ) 个位置上的元素,这个元素的值也就是第  $i$  个元素到第  $n-1$  个元素中的最小值。选择排序算法的流程图如图 6-46 所示。

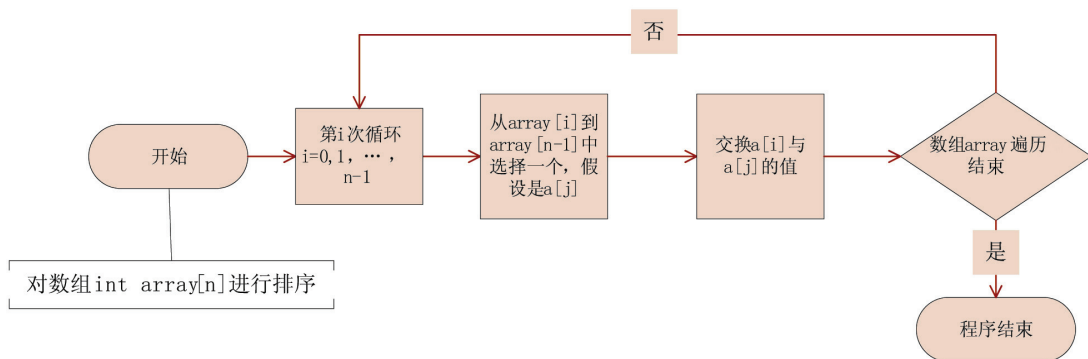


图 6-46 选择排序流程图

#### 【关键代码】

```
int a[10];
int i;
for(i=0;i<10;i++)
{
    cin>>a[i];
}
sort(a,10);
cout<<"排序后的结果为: "<<endl;
for(i=0;i<10;i++)
```



```

    {
        cout<<a[i]<<" ";
    }
void sort(int a[],int n)
{
    int i=0,k=0;
    int temp;
    for(i=0;i<n;i++)
    {
        for(k=i+1;k<n;k++)
        {
            if(a[k]<a[i])
            {
                temp=a[i];
                a[i]=a[k];
                a[k]=temp;
            }
        }
    }
}

```

【题目 6-5】Fibonacci 数列本身就是用递归定义的，适合用递归函数来解决。要求：编写一个函数，求该数列的第  $n$  个是多少，输入的测试数据要有多组。程序的运行效果如图 6-47 所示。

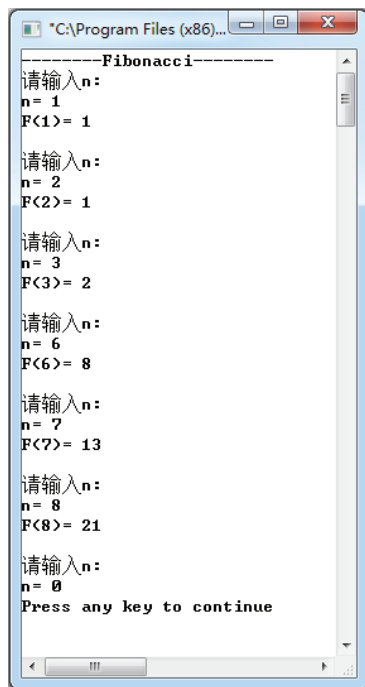


图 6-47 运行效果

在 Fibonacci 数列中，第一个和第二个数都是 1，其后的每一个数都是这个数前面两个数的和。

【题目分析】本题考查递归函数的应用。在 Fibonacci 数列中，如果要求第  $n$  个数，则首先应当求出第  $(n-2)$  和第  $(n-1)$  个数。为了求出第  $(n-2)$  和第  $(n-1)$  个数，又需要求出第  $(n-3)$  和第  $(n-4)$  个数……如此递归下去，直至第 1 和第 2 个数。而第 1 个数和第 2 个数是已知的，递归到此终止。递归终止以后，就开始“反递归”，即根据递归终止时求得的值，



一步步返回去，求得目标的值。编程时只要编写递归过程即可。在函数 `Fibonacci` 的定义中，“`1==n||2==n`”就是递归的终止条件。若这个条件不满足，则继续递归。

### 【关键代码】

```
int main()
{
    int Fibonacci(int);
    cout<<"-----Fibonacci-----"<<endl;
    int n=0;
    cout<<"请输入 n: "<<endl;
    cout<<"n= ";
    cin>>n;
    while(0<n)
    {
        cout<<"F("<<n<<")= "<<
            Fibonacci(n)<<endl<<endl;
        cout<<"请输入 n: "<<endl;
        cout<<"n= ";
        cin>>n;
    };

    return 0;
}

int Fibonacci(int n)
{
    if(1==n||2==n)
    {
        return 1;
    }
    return Fibonacci(n-1)+Fibonacci(n-2);
}
```

**【题目 6-6】**编写一个判断素数的函数，在主函数中调用它。要求：让用户输入数据时要有提示信息，并且输入的数据不能只有一组。也就是说，数据输入的结束要让用户来决定。程序的运行效果如图 6-48 所示。

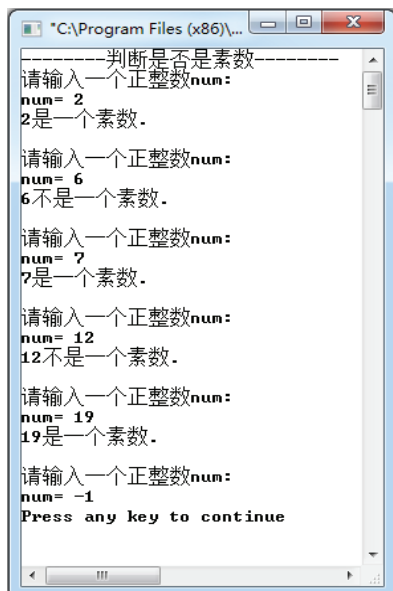


图 6-48 运行结果



【题目分析】本题考查函数定义、函数调用的相关知识。

【关键代码】

```
bool IsprimeNumber(int);
cout<<"-----判断是否是素数-----"<<endl;

int num=0;
cout<<"请输入一个正整数 num: "<<endl;
cout<<"num= ";
cin>>num;
while(0<num)
{
    if(IsprimeNumber(num))
    {
        cout<<num<<"是一个素数."<<endl;
    }
    else
    {
        cout<<num<<"不是一个素数."<<endl;
    }
    cout<<endl<<"请输入一个正整数 num: "<<endl;
    cout<<"num= ";
    cin>>num;
}
bool IsprimeNumber(int num)
{

    int s=0;
    if(1==num||2==num)
    {
        return true;
    }

    s=static_cast<int>(sqrt(num));
    int i=0;
    for(i=2;i<=s;i++)
    {
        if((num%i)==0)
        {
            return false;
        }
    }
    return true;
}
```

# 第 7 章 指针与引用

指针和引用被认为是 C++ 中的特殊数据类型，它与前面章节介绍的基本数据类型不同。使用指针和引用可使程序简洁、紧凑和高效，所以对于每一个学习 C++ 语言的人，都要掌握指针和引用的使用方法。指针和引用的用法比较特殊，而且运用非常灵活。本章将详细讲述指针和引用的概念和使用。

## 7.1 指针概述

计算机的数据都是存储在内存中的，内存是按字节排列的存储空间，每个字节都有一个编号，被称为地址，程序中用到的数据和声明的变量都存放在这一个个字节中，不同类型的数据和变量占用的字节数不同，习惯上将某个变量占用的字节数称为内存单元，指针就是记录这些地址的变量，而指针的类型表示指针指向地址存储的数据类型。准确地理解指针的概念是掌握指针的前提。

### 7.1.1 指针的基本概念

指针是一个地址，它指向存储某一个数据的存储地址。此外，还有一个指针变量的概念，如图 7-1 所示。

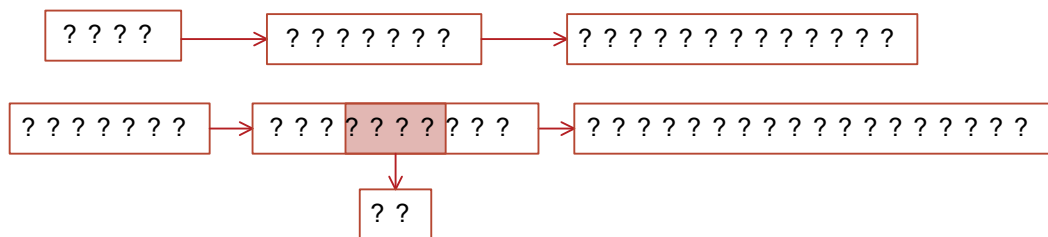


图 7-1 指针变量及其访问方法

在现实生活中，指针的概念也是比较常见的。例如，高速公路上的交通指示牌指示了某地的地理位置，这就是指针，而这个指示牌就是指针变量，用于存储指针，如图 7-2 所示。

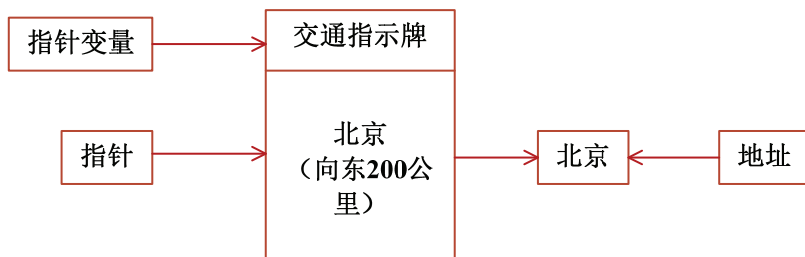


图 7-2 指针的概念

在 C++ 中，如果在内存中存储了一个变量 *a*，其值为 100，那么通过指针变量 *p* 访问该变量的流程如图 7-3 所示。

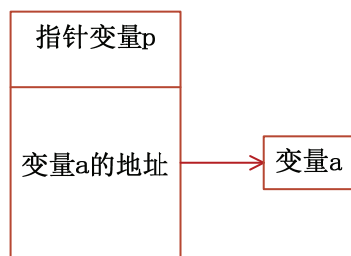


图 7-3 指针

可以看到，指针变量 *p* 指向变量 *a*。在理解“指向”时，应该了解指针变量 *p* 中存有变量 *a* 的地址，通过该地址就能找到变量 *a*。因此，在 C++ 语言中用指针来表示指向关系，即指针就是地址。



**注意：**在 C++ 具体程序中参加数据处理的量不是指针本身的量，而是指针所指向的变量，即指针所指向的内存区域中的数据（称为指针的目标）才是需要处理的数据。

### 7.1.2 定义指针变量

指针是一种复合型的数据类型，基于该类型声明的变量称为指针变量，该变量存放在内存中的某个地址，与其他基本数据类型一样，使用指针之前也必须先定义指针变量。在 C++ 中，定义指针变量的一般形式如图 7-4 所示。

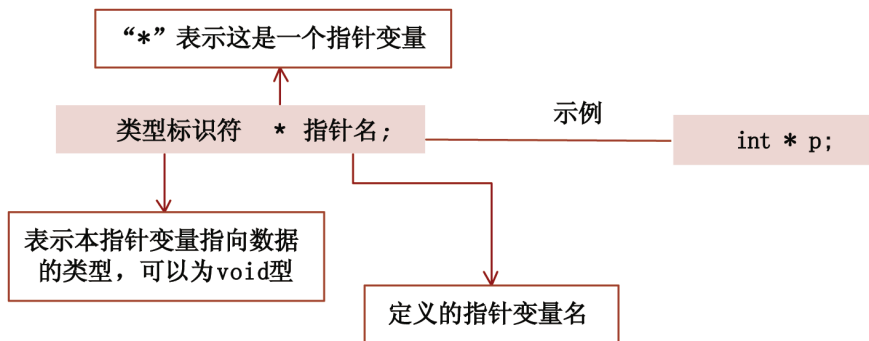


图 7-4 定义指针变量的一般形式



需要注意的是，定义一个指针变量必须用符号“\*”，它表明其后的变量是指针变量，但不要认为“\*p”是指针变量，指针变量是 p 而不是 \*p。

此外，有相同存储类型和数据类型的指针可以在一行中说明，也可以和同类型的普通变量在一起说明，如图 7-5 所示。

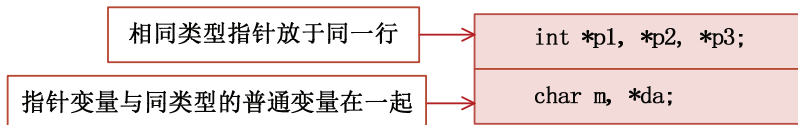


图 7-5 指针变量的说明

当在一行中定义多个同一类型的指针时，用逗号隔开各指针变量标识符，并且每个变量前都要加上“\*”。否则，该变量就不是一个指针，而是一个普通类型的变量。如图 7-6 中定义的变量语句。



图 7-6 不加“\*”含义不同



**注意：**定义指针变量时，“\*”可以出现在类型名和变量名之间的任何位置，例如：

`int *p,q;`

等价于：

`int* p,q;`

p 是整型指针变量，q 是整型变量。第二种写法容易理解为 p、q 都是指针变量。故建议写成第一种形式。

指针变量存储的内容是内存中某个字节的地址，指针变量占用的内存字节数随系统的不同而不同。

### 7.1.3 初始化指针

定义了一个指针后，在使用此指针前，必须给它赋一个合法的值。在 C++ 中，可以在定义指针的同时通过初始化来给指针赋值，也可以在使用之前给指针赋值。

一般来说，C++ 在定义指针的同时初始化指针，其形式如图 7-7 所示。

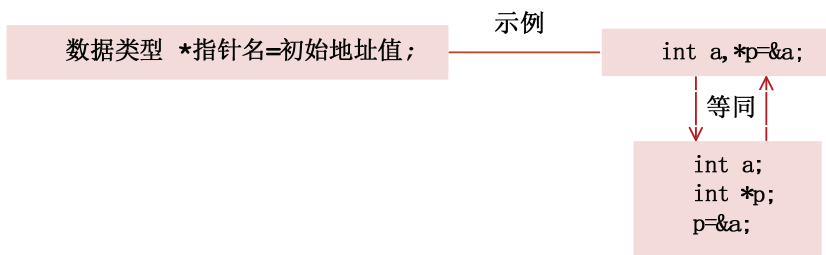


图 7-7 C++ 在定义指针的同时初始化指针的形式





**注意：**当把一个变量的内存地址作为初始值赋给指针时，该变量必须在指针初始化之前已做说明，因为变量只有在说明之后才被分配一定的内存地址。此外，该变量的数据类型必须与指针的数据类型一致，因此不能将一个浮点型变量的地址赋值给整型的指针变量。

同时，也可以向一个指针赋初值作为另一个指针变量，即把另一个已经初始化的指针赋给一个指针。此时，这两个指针指向同一变量的内存地址，如图 7-8 所示。

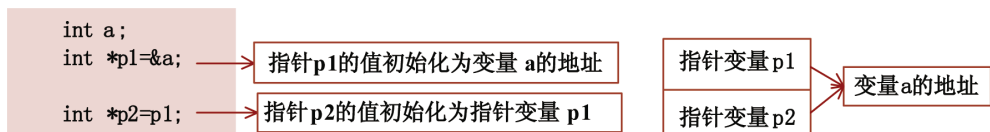


图 7-8 两个指针指向同一变量的内存地址

指针的初始化在具体程序中非常重要，一旦指针的初始化出现问题，就可能导致程序崩溃。

【示例 7-1】下面的程序定义了整型指针变量 `p1` 和 `p2`，并为其赋初值，其实现代码及结果如图 7-9 所示。

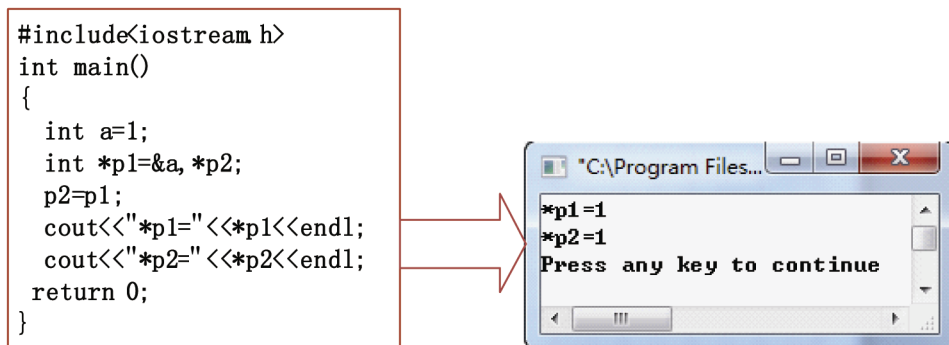


图 7-9 指针的初始化实例



**注意：**在给指针变量初始化时，不能为其赋一个常量，否则程序将通不过编译。

此外，上述程序中使用 `*p1` 和 `*p2` 取出其中存储的变量 `a` 的值，这是下面将要介绍的指针的访问问题。



## 7.2 指针的访问

如果定义一个指针，并使其值为某个变量的地址，则可以通过这个指针间接地访问在这个地址中存储的值。如【示例 7-1】中，通过“\*”符号取出存储在该地址中的值。事实上，在 C++ 中有两个有关指针访问的运算符，下面将依次进行说明。



## 7.2.1 指针的值

指针的值是一个地址。在 C++ 中，为了取得一个变量的地址，引入了取地址运算符 “&” 来取得一个变量的地址，其语法如图 7-10 所示。

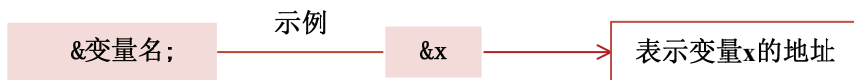


图 7-10 取地址运算符&



**注意：**取地址运算符 “&” 只能应用于内存中存在的数 据，如变量、数组元素等，不能用于表达式、常数或寄存器变量。

【示例 7-2】下面的代码定义了两个指针变量 p1 和 p2，其都指向整型变量 a 后将值分别输出，其实现代码及结果如图 7-11 所示。

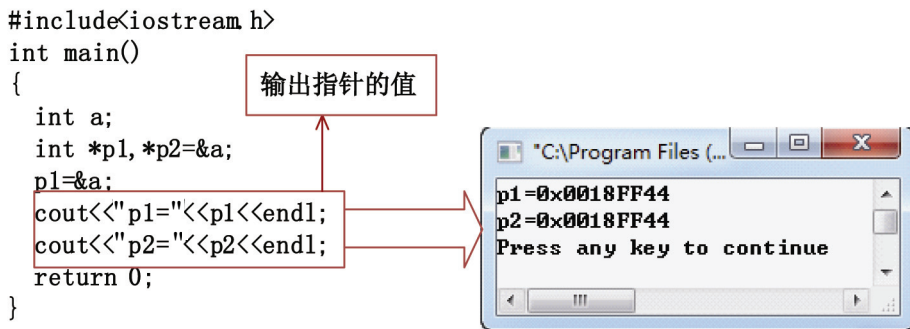


图 7-11 指针的值实例

上述代码在指针的定义中，指针变量是 p1 和 p2 而不是 \*p1 和 \*p2，因此要对一个指针赋值，等号左边不应该加 “\*” 号。



**注意：**C++ 中不允许把一个无效的地址，比如数字赋给指针。指针变量和一般的变量是类似的，存放的值是可以改变的。

## 7.2.2 访问指针数据

通过 “&” 运算符可以获取变量的地址，可将其赋值给指针，即完成了将指针指向该变量的操作。而 C++ 可以通过 “\*” 运算符将指针指向的变量值取出，进行各种运算。

在 C++ 中，“\*” 运算符为取值运算符，也称为指针运算符、指向运算符或间接运算符，\*p 代表 p 所指向的变量。

【示例 7-3】下面的程序接收用户从键盘输入的两个整数，并通过 “\*” 运算符将指针变量 p1 和 p2 的值取出并输出，其实现代码及结果如图 7-12 所示。

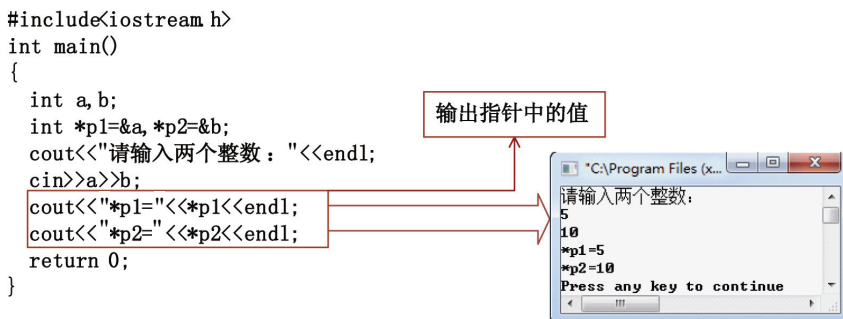


图 7-12 访问指针数据实例

通过指针来访问变量是一种间接的方式，其速度略慢于直接访问。但给程序的开发带来了很大的灵活性，其原因在于，指针也是一个变量，可以在运行时修改其指向，从而达到“使用一个指针，访问多个变量”的目的。

### 7.2.3 小结指针 p

由于引进了指针的概念，在程序中要注意区分下面 3 种表示方法具有的不同意义。例如，有一个指针 p，其不同格式代表的意义不同，如图 7-13 所示。

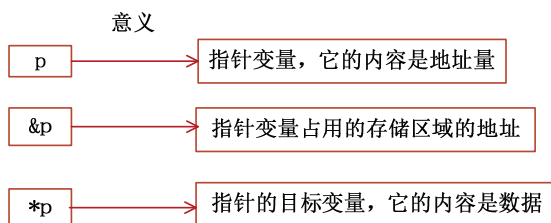


图 7-13 指针 p 不同格式代表的意义

【示例 7-4】下面的程序接收用户输入的两个整数后，分别输出 p、\*p 和&p 的值，实现代码及结果如图 7-14 所示。

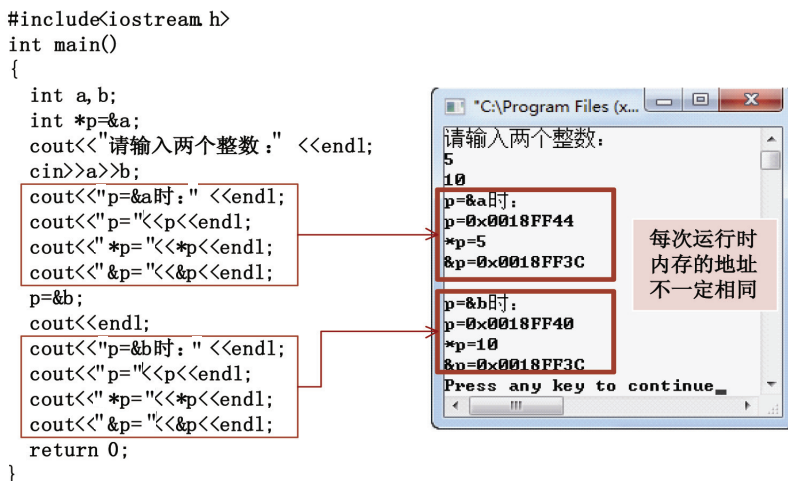


图 7-14 指针 p 格式不同实例



从程序运行结果可以看出，只有 $*p$  的值为  $n$  的值，即其内容为有意义的数据。而  $p$  的值为存储变量  $a$  或  $b$  的地址， $\&p$  的值为指针变量  $p$  的存储地址，这些都是由计算机随机分配的，没有实际意义。因此，在实际程序中，一定要清楚如何使用指针访问数据。

## 7.3 指针的算术运算

指针变量也有加减运算，它可以与某个整型数相加减，也可以与指针相减。但指针与指针相加或相乘除都是没有意义的。指针的值是一个内存地址，而一个内存地址可以用一个整型数表示。因此，指针的算术运算可以看做是整型数之间的一个运算。

### 7.3.1 指针与整数的加减运算

指针与整数的加减运算是指将指针作为地址量加上或减去一个整数  $n$ ，其意义及效果如图 7-15 所示。

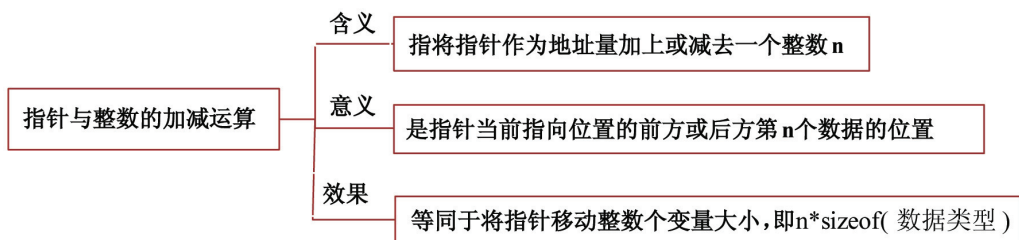


图 7-15 指针与整数的加减运算

由于指针可以指向不同数据类型，即长度不同的数据，所以这种运算的结果取决于指针指向的数据类型。例如，一个 `int` 型（4 字节）指针加减整型数的运算如图 7-16 所示。

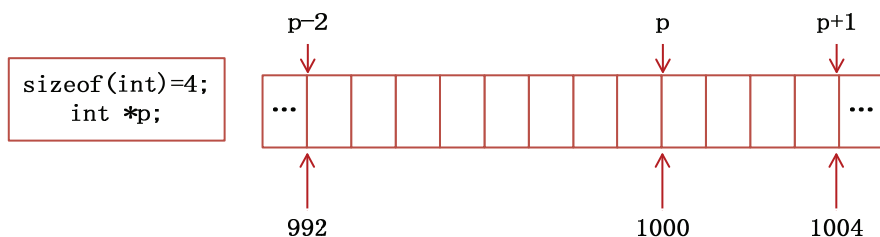


图 7-16 `int` 型指针加减整数的运算

因此，对于某种数据类型的指针  $p$  来说，其实际操作如图 7-17 所示。

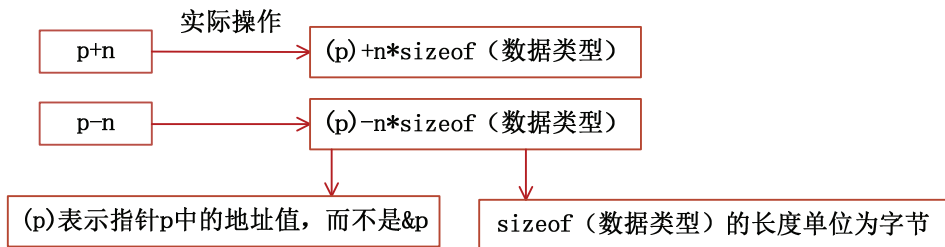


图 7-17 某种数据类型的指针  $p$  的实际操作



### 7.3.2 指针加减 1 运算

指针与 1 的加减运算是一种特殊的指针与整数的加减运算。由于指针的加减 1 运算在具体程序中使用广泛，并有自己的表示方法，因此本小结将单独讲解。

同样地，指针加减 1 单项运算也是地址计算，它具有指针与整数加减运算的特点，指针的加 1、减 1 单项运算是指针中地址值的变化。在 C++ 中，指针  $p$  的加 1、减 1 运算表示如图 7-18 所示。

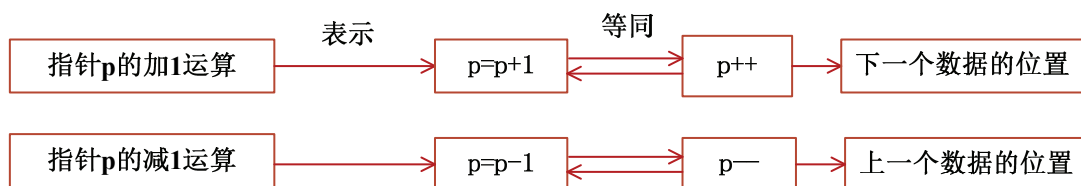


图 7-18 指针  $p$  的加 1、减 1 运算表示

运算后指针地址值的变化量取决于它指向的数据类型。例如，一个 `int` 型指针  $p$  存放的地址为 1000，当执行  $p++$ 、 $p--$  后其地址变化如图 7-19 所示。

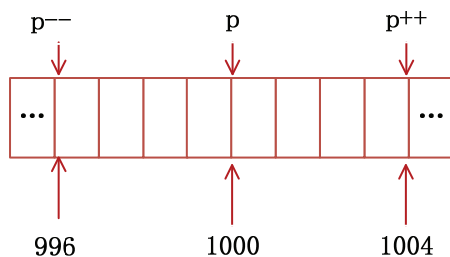


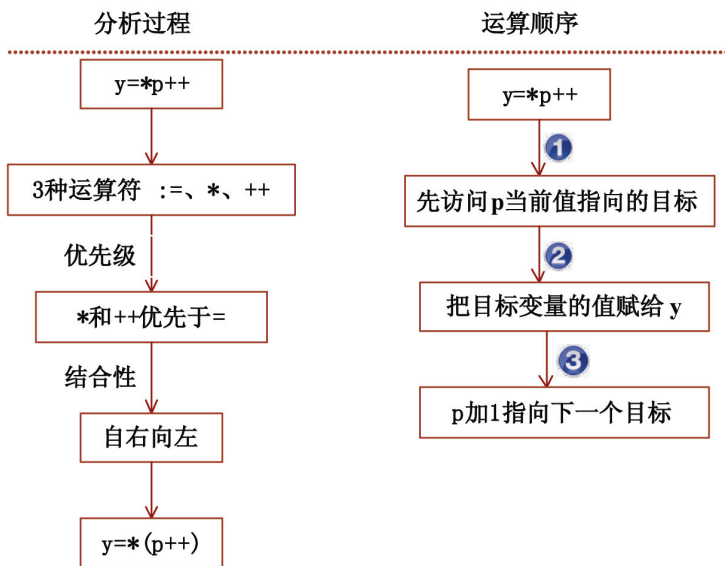
图 7-19 执行  $p++$ 、 $p--$  后其地址变化

指针加 1、减 1 单项运算与前面所讲的自增自减运算符类似，也分为前置运算和后置运算，如图 7-20 所示。



图 7-20 指针加 1、减 1 单项运算分类

当指针加 1、减 1 运算和其他运算出现在一个表达式中时，要注意它们之间的结合规则和运算顺序。例如，表达式  $y=*p++$  的分析过程和运算顺序如图 7-21 所示。

图 7-21 表达式 `y=*p++` 的分析过程和运算顺序

### 7.3.3 指针的相减运算

指针的相减运算是指两个指针所指向的变量类型相同时可以进行减法运算。其运算结果是两个地址之间可存放的变量个数，而不是地址量。例如，两个 `int` 型指针 `px` 和 `py` 进行相减运算，如图 7-22 所示。

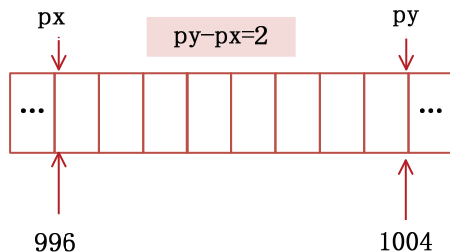


图 7-22 指针的相减运算



**注意：**指针变量也可以进行关系运算，用于比较两个指针是否相等。指针也可以赋值给相同类型的指针变量。



## 7.4 特殊指针

前面学习了数组和函数，C++ 允许指针指向数组和函数。本节将介绍几种特殊的指针。

### 7.4.1 数组指针

在 C++ 中，数组指针是一个指向数组的指针，其定义的一般形式如图 7-23 所示。

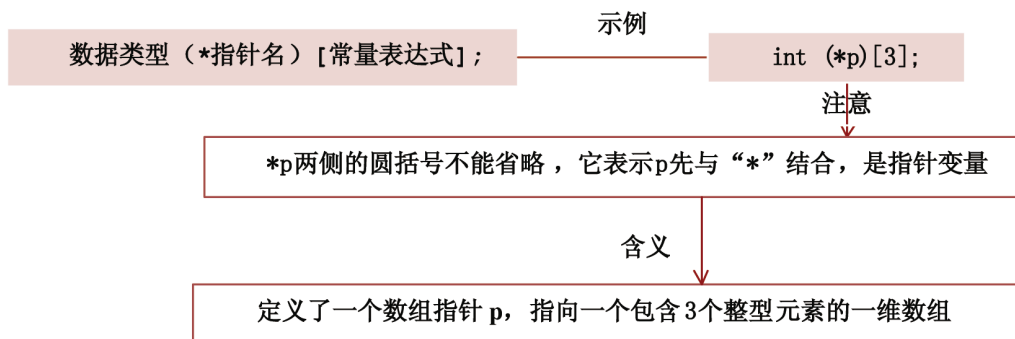


图 7-23 数组指针

【示例 7-7】下列程序定义了一个数组指针，并通过该指针指向某一整型数组，输出其中所有数组元素，其实现代码及结果如图 7-24 所示。

```
#include<iostream h>
int main()
{
    int array[10]={3, 7, 4, 1, 2, 6, 5, 8, 9, 10};
    int (*p)[10];
    p=&array;
    for (int i=0;i<10;i++)
        cout<<(*p)[i]<<" ";
    cout<<endl;
    return 0;
}
```

定义数组指针 p

数组指针 p 初始化

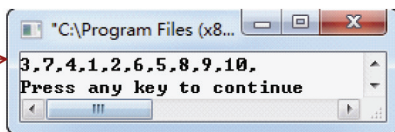


图 7-24 数组指针实例

## 7.4.2 指向函数的指针——函数指针

在 C++ 中，函数指针是一个指向函数的指针，即指针存储的是函数的首地址。其定义的一般形式如图 7-25 所示。

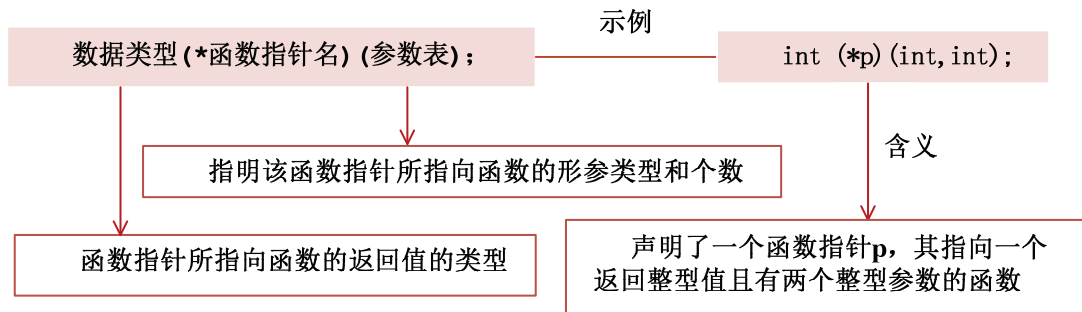


图 7-25 函数指针

在定义了指向函数的指针变量后，在使用此函数指针之前，必须先给它赋值，使它指向一



个函数的入口地址。由于函数名是函数在内存中的首地址，因此可以将函数名赋给函数指针变量，赋值的一般语法格式如图 7-26 所示。



图 7-26 函数指针赋值

【示例 7-8】下面程序定义一个函数指针 p，该指针指向实现两个整型值交换的 swap()函数。在使用该函数指针 p 前为其赋值，并在 main()函数中调用了该函数指针，实现代码及结果如图 7-27 所示。

```
#include<iostream h>
void swap(int *,int *);
int main()
{
    void (*p)(int *,int *);
    p=swap;
    int a,b;
    cout<<"请输入两个整数："<<endl;
    cin>>a>>b;
    cout<<"交换前："<<endl;
    cout<<a<<"\t"<<b<<endl;
    (*p)(&a,&b);
    cout<<endl;
    cout<<"交换后："<<endl;
    cout<<a<<"\t"<<b<<endl;
    return 0;
}
void swap( int *a, int *b )
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

void (\*p)(int \*,int \*);  
p=swap;

cout<<"交换前："<<endl;

cout<<a<<"\t"<<b<<endl;

(\*p)(&a,&b);

cout<<endl;

cout<<"交换后："<<endl;

cout<<a<<"\t"<<b<<endl;

return 0;

}

void swap( int \*a, int \*b )

{

int temp = \*a;

\*a = \*b;

\*b = temp;

}

定义函数指针 p，并初始化

cout<<"交换前："<<endl;

cout<<a<<"\t"<<b<<endl;

(\*p)(&a,&b);

cout<<endl;

cout<<"交换后："<<endl;

cout<<a<<"\t"<<b<<endl;

return 0;

}

void swap( int \*a, int \*b )

{

int temp = \*a;

\*a = \*b;

\*b = temp;

}

```
*C:\Program Files (x...
请输入两个整数:
5
10
交换前:
5      10
交换后:
10     5
Press any key to continue_
```

图 7-27 函数指针实例

### 7.4.3 指针数组

指针数组就是其元素为指针的数组，如图 7-28 所示。

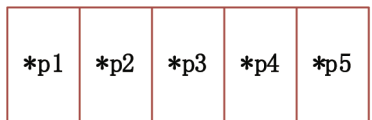


图 7-28 指针数组

指针数组是指针的集合，其中的每一个元素都是指针变量，并且它们具有相同的存储类型，





指向相同的数据类型。指针数组在使用前必须先声明，C++中声明指针数组的一般形式如图7-29所示。

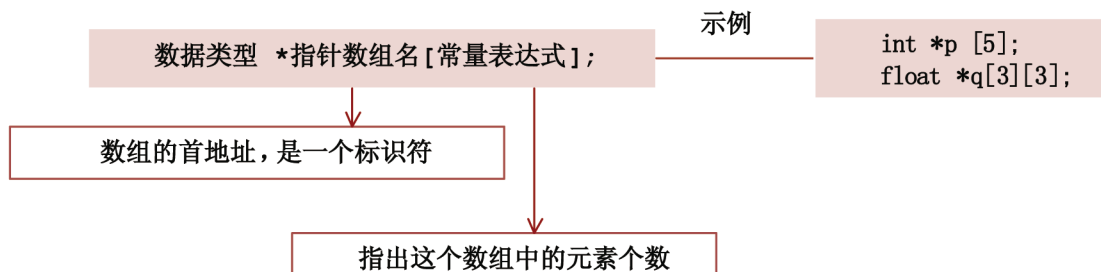


图 7-29 指针数组的声明

与普通指针类似，指针数组在使用前也必须先赋值，否则指针可能指向没有意义的值。指针数组赋初值与一般数组的赋值类似，可以在声明指针数组的同时进行初始化。

【示例 7-9】下面的程序定义了一个指向字符串的包含 5 个元素的指针数组，初始化后将其倒序输出，其实现代码及结果如图 7-30 所示。

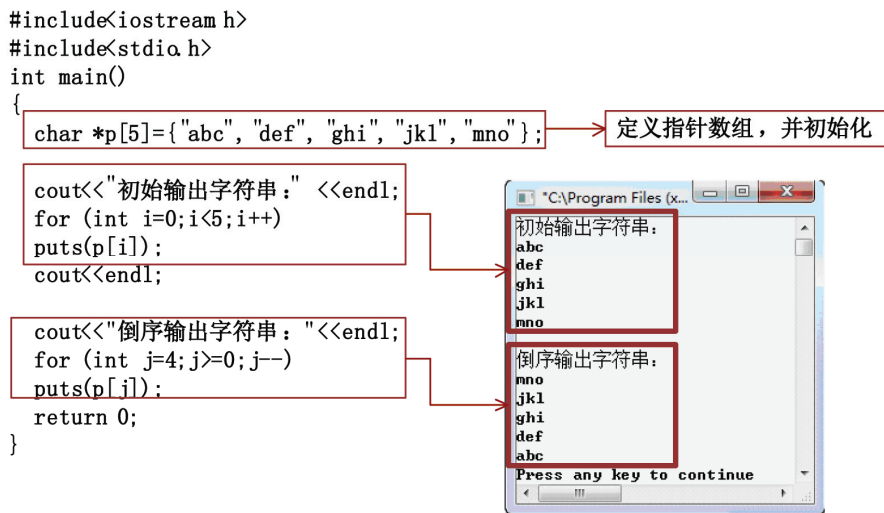


图 7-30 指针数组实例



**说明：**要严格区分数组指针和指针数组。

#### 7.4.4 二级指针——指针的指针

虽然指针存储的是一个地址，但指针本身也是一个变量，在内存中占据一定的空间，并且具有一个地址，这个地址也可以利用指针来保存。因此，同样可以声明一个指针来指向它，这个指针称为指向指针的指针。

在 C++ 中，指向指针的指针也被称为二级指针，在声明指向指针的指针时，其形式与声明指针类似，但需加上两个间接取值的运算符“\*”，声明形式如图 7-31 所示。

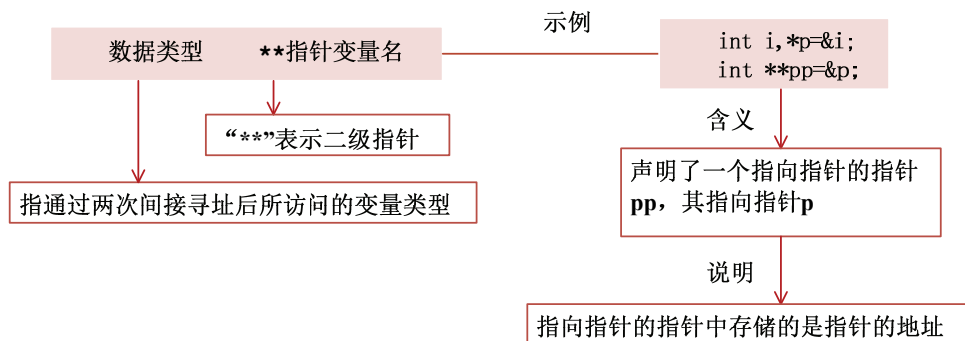


图 7-31 二级指针的声明与定义

【示例 7-10】下面的程序声明了一个指针 **p**，一个指向指针的指针 **pp**，将指针 **pp** 指向指针 **p**，实现代码及输出结果如图 7-32 所示。

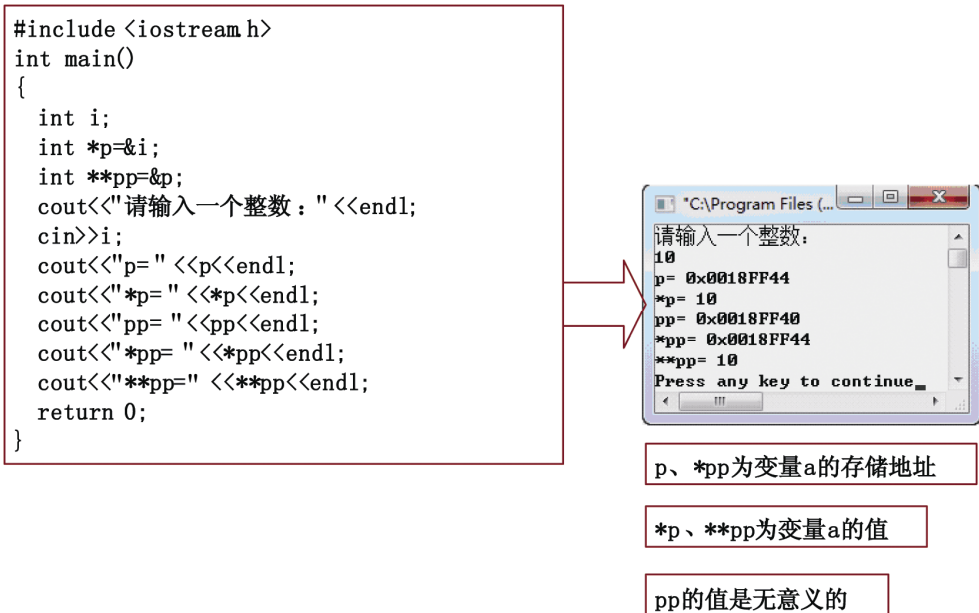


图 7-32 二级指针实例

### 7.4.5 多级指针——二级以上的指针

多级指针是指含有多个间接取值运算符“\*”的指针，其声明形式与二级指针类似，如图 7-33 所示。

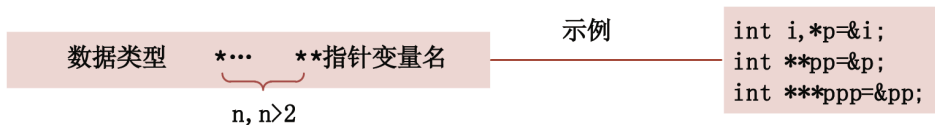


图 7-33 多级指针的声明

在 C++ 中常用的是二级指针，多级指针只需了即可，在这里不做详细讲解。



## 7.5 指针的应用

C++中,使用指针可使程序简单、可读性强,并且指针的使用非常灵活。本节将重点讲解指针在数组、字符串、函数中的应用及动态内存分配。

### 7.5.1 指向一维数组的指针

任何数据类型中的数组元素,除了用数组名下标的方法进行访问外,还可以用指针访问。用指针访问数组形式简单、使用灵活,程序的可读性强。

#### 1. 指针访问数组元素

用指针指向数组就是让指针指向这段连续内存的首地址,即数组中第一个元素(下标为0)的地址。定义一个指向数组的指针变量,只要其与数组元素类型相同即可。

由于数组是一段连续的内存,指针可以指向数组,而且可以通过加、减整数来移动指针。所以,可以通过指针来访问数组,即数组中的元素。

使用指针访问数组,同用下标访问数组的效果是一样的。例如,一个指向数组 `arr` 首地址的指针 `p`,访问第 `i+1` 个元素(下标为 `i`),可以用 `*(p+i)`,也可以用 `arr[i]`,这两种方法是等价的。由于数组名代表的是数组的首地址,所以也可以用 `*(arr+i)` 来访问第 `i+1` 个元素,如图 7-34 所示。

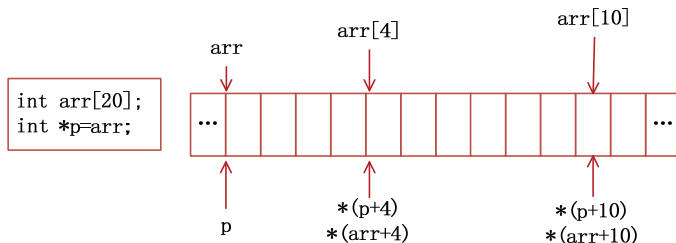


图 7-34 指针访问数组元素

【示例 7-11】下面的程序通过几种方式对数组元素进行访问,实现代码及结果如图 7-35 所示。

```
#include<iostream>
int main()
{
    int array[10]={3,7,4,1,2,6,5,8,9,10};
    int *p;
    p=array;
    int i;
    cout<<"下标访问数组元素: "<<endl;
    for (i=0;i<10;i++)
        cout<<array[i]<<" ";
    cout<<endl<<endl;

    cout<<"地址访问数组元素: "<<endl;
    for (i=0;i<10;i++)
        cout<<*(array+i)<<" ";
    cout<<endl<<endl;

    cout<<"指针访问数组元素: "<<endl;
    for (i=0;i<10;i++)
        cout<<*(p+i)<<" ";
    cout<<endl<<endl;

    return 0;
}
```

定义指针变量并初始化

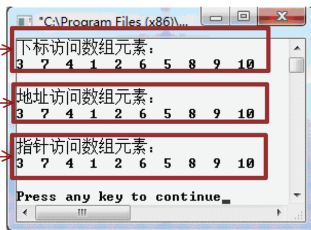


图 7-35 几种方式对数组元素进行访问



无论是采用下标、地址，还是指针都可以得到相同的访问结果。但是，在数组元素的访问中，使用指针进行访问更为灵活。

## 2. 指向一维数组

在实际程序中一维数组的使用最为频繁，因此使用指针指向一维数组是具体应用中使用最多的。

【示例 7-12】下面的程序将一个数组的元素进行反转，即第一个元素放到最后一个，第二个元素放到倒数第二个，以此类推，其实现代码及结果如图 7-36 所示。

```
#include<iostream h>
int main()
{
    int array[10]={3,7,4,1,2,6,5,8,9,10};
    int i;
    cout<<"反转前: "<<endl;
    for( i=0; i<10; i++ )
    {
        cout<<array[i]<<" ";
    }
    cout<<endl<<endl;
    int *p = array;
    int *q = p +9;
    while( p < q )
    {
        int t = *p;
        *p = *q;
        *q = t;
        p++;
        q--;
    }
    cout<<"反转后: "<<endl;
    for( i=0; i<10; i++ )
    {
        cout<<array[i]<<" ";
    }
    cout<<endl<<endl;
    return 0;
}
```

指向数组头的指针

指向数组尾的指针

只要两个指针不相遇，就交换  
两个指针所指元素的值

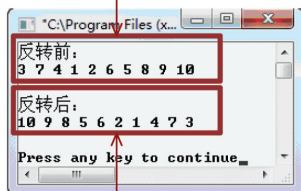


图 7-36 指向一维数组的实例

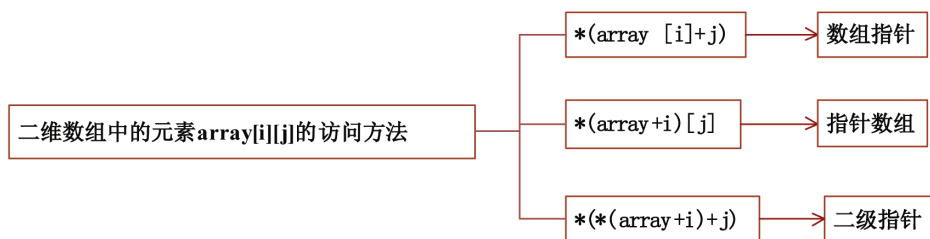


**注意：**在使用指针访问数组时不要越界，即保证指针指向数组第一个到最后一个元素。

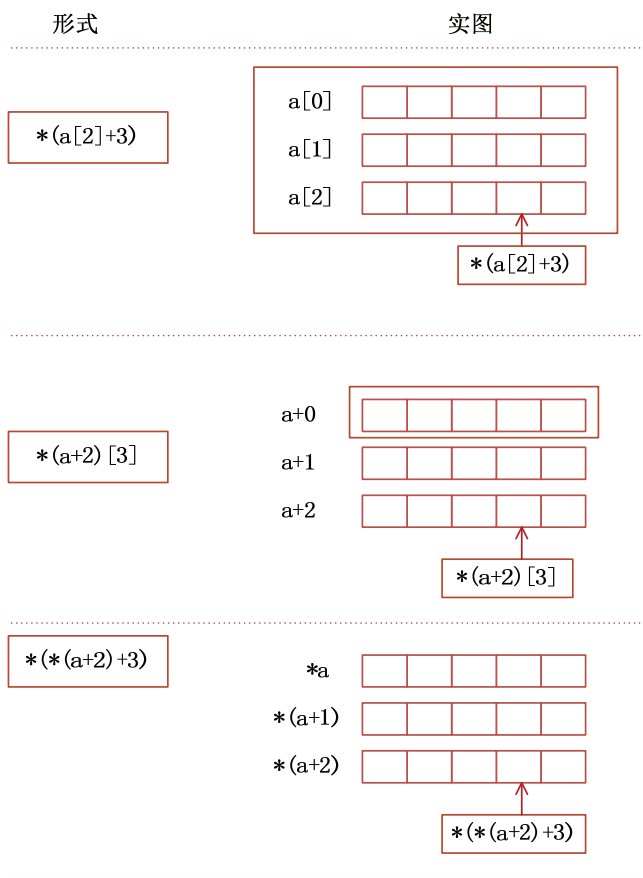
## 7.5.2 指向二维数组的指针

多维数组，尤其是二维数组在具体程序中的应用非常广泛，通过指针来访问二维数组元素也是常用的。

对于一维数组 `array[10]`而言，指针指向数组的首地址，即`&array[0]`的值。而对于二维数组 `array[10][10]`而言，数组的首地址为`&array[0][0]`的值。因此，对于二维数组中的元素 `array[i][j]`有多种访问方法，如图 7-37 所示。

图 7-37 二维数组中的元素  $\text{array}[i][j]$  的访问方法

例如，一个数组  $\text{a}[3][5]$  中元素  $\text{a}[2][3]$  的 3 种等价访问方法如图 7-38 所示。

图 7-38 元素  $\text{a}[2][3]$  的 3 种等价访问方法

二维数组  $\text{a}[3][5]$  是一个  $3 \times 5$  的矩阵，包括 3 行，每一行都有起始地址。C++ 中以  $\text{a}[0]$ 、 $\text{a}[1]$ 、 $\text{a}[2]$  分别表示第 0 行、第 1 行、第 2 行的起始地址，即该行第 0 列元素的地址。



**注意：**二维数组的  $\text{a}[0]$ 、 $\text{a}[1]$ 、 $\text{a}[2]$  并不是一个元素，而是一行首地址，正如同一维数组名是数组起始地址一样。因此， $\text{a}[0]$  的值等于  $\&\text{a}[0][0]$ ， $\text{a}[1]$  的值等于  $\&\text{a}[1][0]$ ， $\text{a}[2]$  的值等于  $\&\text{a}[2][0]$ 。



【示例 7-13】下面的程序指向二维数组 `array` 首元素的指针 `p`，通过几种指针访问元素的方式将数组中的元素依次输出，其实现代码及结果如图 7-39 所示。

```
#include<iostream h>
int main()
{
    int array[3][3]={3, 7, 1, 2, 6, 5, 8, 9, 10};
    int i, j;
    int (*p)[3]=array;
    for ( i=0;i<3;i++)
    {
        {
            for ( j=0;j<3;j++)
                cout<<p[i][j]<<" ";
        }
        cout<<endl;
    }
    cout<<endl;
    for ( i=0;i<3;i++)
    {
        {
            for ( j=0;j<3;j++)
                cout<<*(p[i]+j)<<" ";
        }
        cout<<endl;
    }
    cout<<endl;
    for ( i=0;i<3;i++)
    {
        {
            for ( j=0;j<3;j++)
                cout<<*(*(p+i)+j)<<" ";
        }
        cout<<endl;
    }
    cout<<endl;
    return 0;
}
```

定义数组指针并初始化

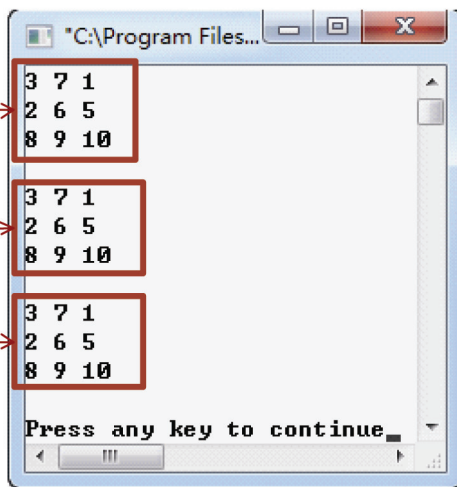


图 7-39 指向二维数组指针实例

### 7.5.3 指向字符串的指针

在 C++ 中，字符串是用字符数组表示和存储的。数组的访问能够通过指针来实现，因此字符串也同样可以通过指针来访问。

指向字符串的指针就是一个 `char` 类型的指针。与普通指针一样，字符串指针在使用前也必须先定义。例如，下面的语句定义了一个字符串 `str`，并定义了一个指向该字符串的指针 `p`，并对该指针进行初始化，如图 7-40 所示。



图 7-40 指向字符串的指针

【示例 7-14】用字符串指针访问字符串，并使用字符串函数对字符串进行比较和计算长度操作，实现代码及结果如图 7-41 所示。

```
#include<iostream h>
#include<string.h>
int main()
```

```
{
```

```
char *s="Hello ";
char *t="World ";
```

定义字符串指针，并初始化

```
int i=strcmp(s,t);
if (i>0)
cout<<"s>t"<<endl;
else if (i<0)
cout<<"s<t"<<endl;
else
cout<<"s==t"<<endl;
```

比较字符串大小，  
输出结果

```
int j=strlen(s);
cout<<endl;
cout<<"字符串s的长度为："<<j<<endl;
return 0;
```

计算字符串长度，并输出

```
}
```

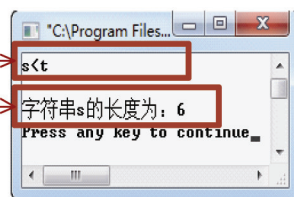


图 7-41 指向字符串指针实例

### 7.5.4 指针作为函数参数

在函数的参数列表中，可以使用指针类型的参数。传递给指针参数的实参可以是一个指针变量，也可以是一个变量的地址。在 C++ 中，使用指针作为参数可以提高传递参数的效率，而且在函数中可以修改实参指针所指变量的值。

例如，下面声明函数 `function()` 时其形式参数就是一个 `int` 类型的指针，在调用函数 `function()` 时必须传入一个 `int` 型的指针变量，如图 7-42 所示。



```
void function ( int * );  
int *p;  
...  
function( p );
```

定义一个int型指针变量

某些操作改变了p的值

以p作为实参，调用函数function()

图 7-42 指针作为函数参数



**注意：**使用指针作为函数的形式参数，在调用该参数时传递的是地址。

【示例 7-15】将指针作为函数的参数进行传递，完成两个数之间的互相交换功能，使用的是地址传递的方式，其实现代码及结果如图 7-43 所示。

```
#include<iostream>  
using namespace std  
int main()  
{  
    void change(int *,int *);  
        //change()函数声明，形参为指针  
    int x=2,y=3;  
    cout<<"交换前: x= "<<x<<", y= "<<y<<endl;  
    change(&x,&y);  
        //change()函数的调用，传递的是地址  
    cout<<"交换后: x= "<<x<<", y= "<<y<<endl;  
    return 0;  
}  
  
void change(int *n,int *m)  
        //change()函数的定义  
{  
    int temp;  
    temp=*n;  
    *n=*m;  
    *m=temp;  
}
```

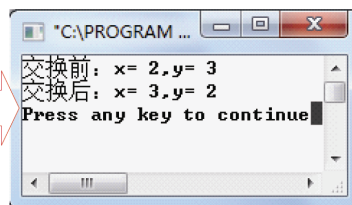


图 7-43 指针作为函数参数实例





代码“change(&x,&y);”中的 change()函数成功地实现了 x 和 y 之间的数据交换,函数的形参是两个 int 型指针,将调用函数中的变量地址作为实参,赋值给形参,完成对调用函数中变量的处理。

### 7.5.5 指针作为函数的返回值——指针函数

指针函数是指函数的返回值为指针类型。指针函数在调用后返回一个指针,通过指针中存储的地址值主调函数就能访问该地址中存放的数据,并通过指针算术运算访问这个地址的前、后内存中的值。

指针函数与一般函数的区别如图 7-44 所示。

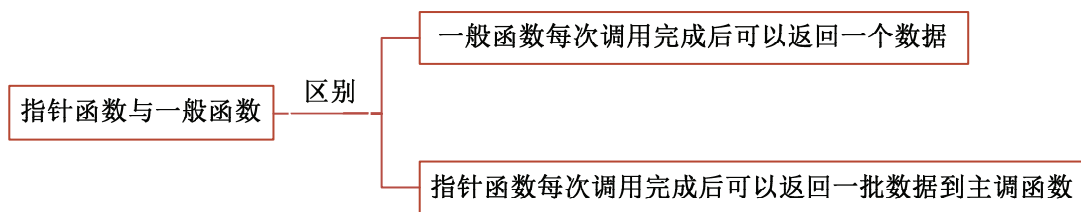


图 7-44 指针函数与一般函数的区别

在 C++ 中,指针函数与一般函数的声明和定义形式类似,不同点在于其返回值必须为指针,其一般语法形式如图 7-45 所示。

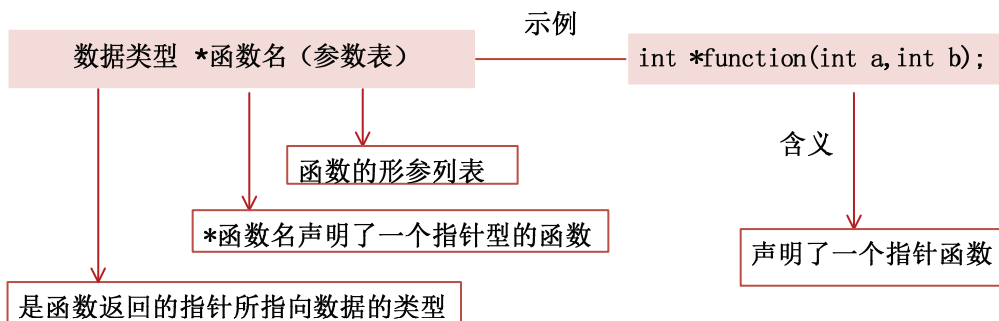


图 7-45 指针函数的声明

在具体程序中,指针函数的优势在于能够返回一组数据,因此指针函数多用于数组和字符串的处理。

【示例 7-16】下面的程序定义了一个包含 5 个字符串的指针数组,将这些字符串进行比较后输出其中最大的一个,其实现代码及结果如图 7-46 所示。

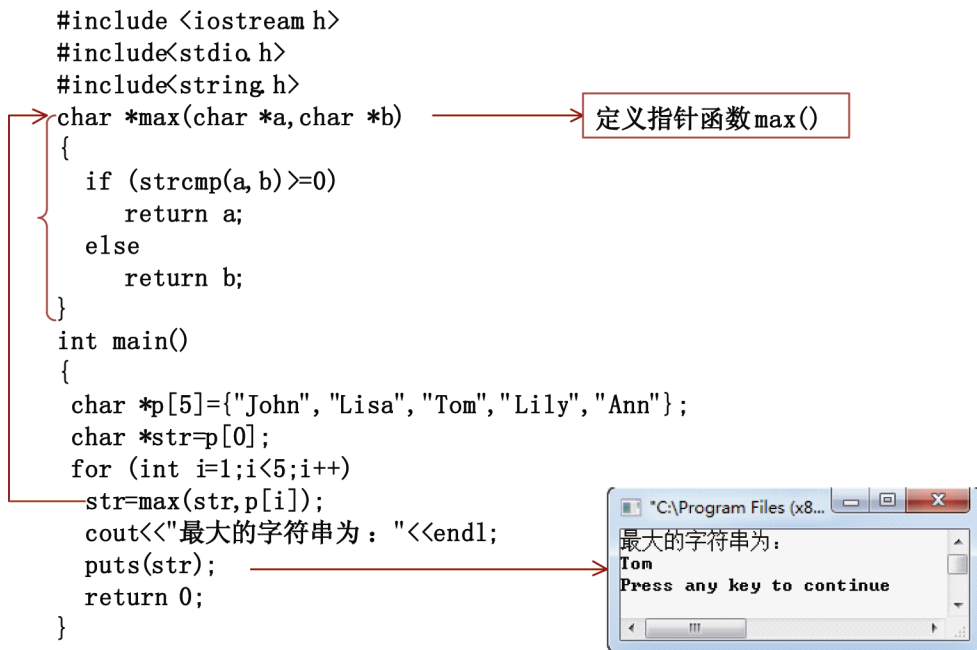


图 7-46 指针函数实例



**注意：**函数指针与指针函数千万不要混淆。

## 7.5.6 动态内存分配

指针存储的是内存地址，在使用指针时，需要保证指向地址的有效性。C++程序中的内存分配分为两种，如图 7-47 所示。

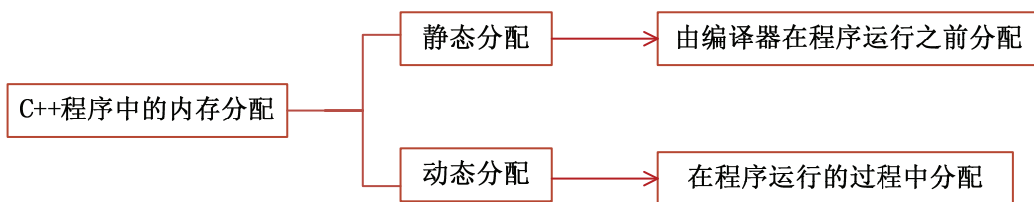


图 7-47 C++程序中的内存分配

在 C++中，通过关键字 `new` 和 `delete` 来实现程序的动态内存分配和回收，如图 7-48 所示。

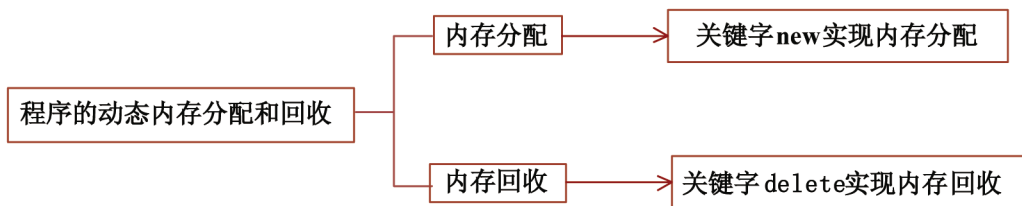


图 7-48 动态内存分配和回收



其中,关键字 `new` 实现内存分配,如果需要对分配出的内存进行初始化,则在类型后面加上一个括号,并带有初始值,其一般形式如图 7-49 所示。

类型标识符 \*指针名 = new 类型标识符(初始值);

示例

```
int * p = new int(10);
```

```
int * p = new int[10];
```

图 7-49 C++中动态分配内存的一般形式

此外,通过关键字 `new` 分配的内存必须由开发者去释放。一块内存如果没有被释放,则会一直存在到该应用程序结束。在 C++中,使用 `delete` 来释放内存,其一般形式如图 7-50 所示。

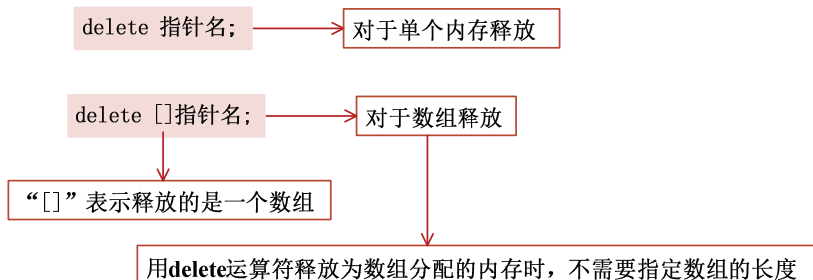


图 7-50 delete 释放内存的一般形式

【示例 7-17】下面的程序为一个整数和一个整型数组动态分配内存空间,使用该空间存储用户输入的数组元素,最后将这些空间释放,实现代码及结果如图 7-51 所示。

```

#include <iostream.h>
int main()
{
    int * p = new int;
    int i;
    cout<<"请输入数组的长度: "<<endl;
    cin>>*p;
    cout<<"请输入数组的元素: "<<endl;
    int * q = new int[*p];

    for( i = 0; i < *p; i ++ )
    {
        cin>>q[i];
    }

    cout<<"数组元素为: "<<endl;
    for( i = 0; i < *p; i ++ )
    {
        cout<<q[i]<<"\t";
    }
    cout<<endl;
    delete p;
    delete []q;
    return 0;
}
    
```

动态创建一个变量

动态创建一个数组

释放变量和数组

图 7-51 动态内存分配实例



**注意：**使用关键字 `new` 分配的内存都需要显式使用关键字 `delete` 释放。如果缺少了 `delete` 语句，则会造成内存泄露，这是非常危险的。

用 `new` 申请动态数组，格式如下：

类型名 \*指针变量名 = `new` 类型名 [元素个数];

其中，元素个数可以是变量。



## 7.6 引用

虽然指针的使用非常灵活和高效，但使用起来却不是非常方便，如果使用不当，很容易导致某些不易察觉的错误。为此，C++引入了引用。

### 7.6.1 引用的应用

在 C++ 中，引用也是一种数据类型。引用不能独立存在，而只能依附于一个变量。所以，定义一个引用必须指明是哪个变量的引用。定义一个引用包括目标变量的数据类型、引用修饰符“&”、引用的标识符及目标变量的标识符，其语法如图 7-52 所示。

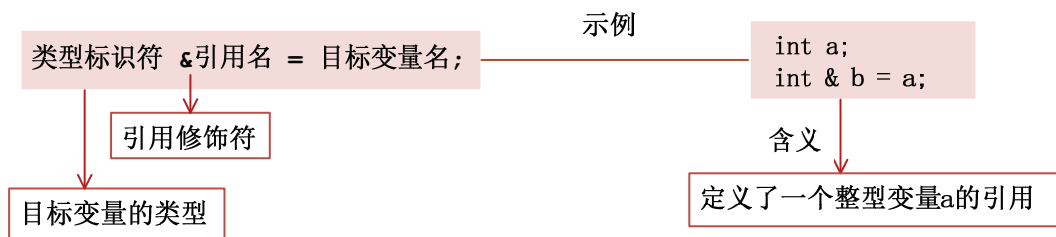


图 7-52 引用的一般语法形式



**注意：**此处的“&”不是取地址运算符，而是一个引用修饰符。

引用一旦定义，即始终跟其目标变量绑定，而不能改变为其他变量的引用。假如 `b` 是变量 `a` 的引用，则在 `b` 的生命周期内，`b` 始终都是 `a` 的引用，而不能再改变为其他变量的引用。

此外，引用在其生命周期内完全可以替代其目标变量。也就是说，所有施加于引用上的操作，其效果等同于直接对引用的目标变量操作。而且一旦目标变量的值发生了改变，引用的值也会发生同样的改变。如图 7-53 所示的程序体现了引用的这个特征。

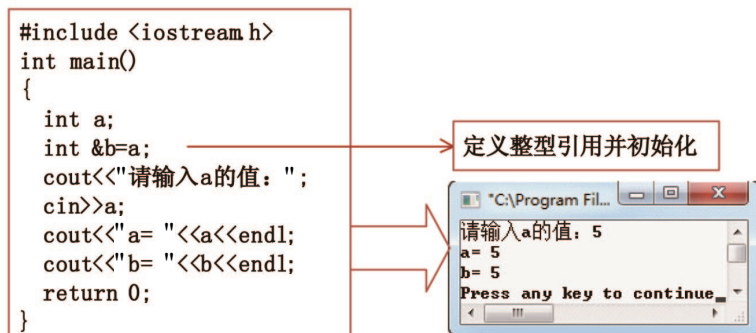


图 7-53 引用的特征

鉴于引用的不可变更性, 以及引用与目标变量的等价性, 一个变量的引用也可以看做是该变量的别名。定义一个引用只不过是给变量另外命名。这样两个名字拥有一个实体, 对一个名字的操作自然也会影响到另外一个名字。

### 7.6.2 引用与指针

与传统的 C 语言相比, 引用是 C++特有的类型。在很多情况下, 引用提供了与指针操作同等的能力, 主要表现如图 7-54 所示。

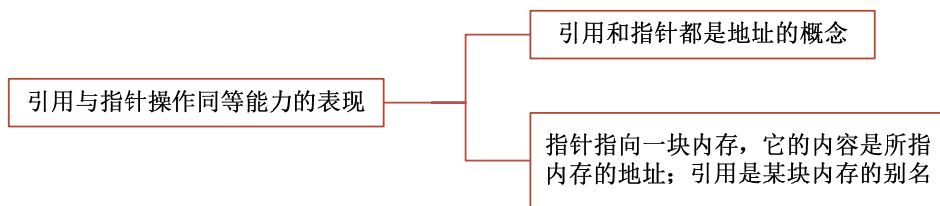


图 7-54 引用与指针操作同等能力的主要表现

C++中指针与引用都是用于间接引用其他对象, 但是它们还是有一些区别, 如图 7-55 所示。

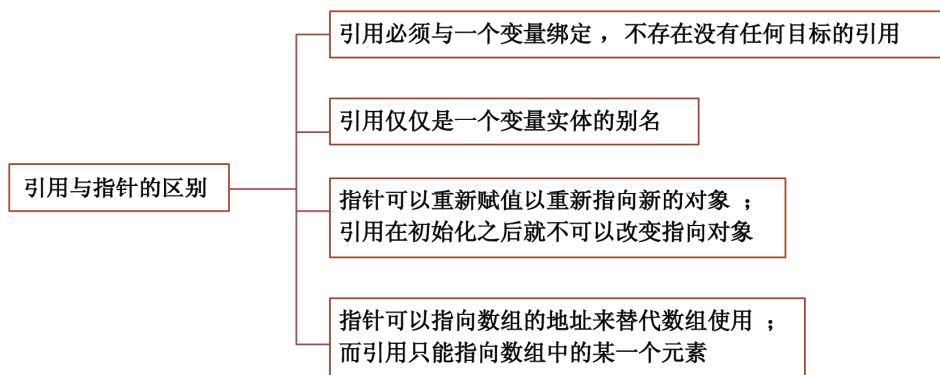


图 7-55 引用与指针的区别



**注意:** 在使用时应当根据实际情况选择指针和引用。



【示例 7-18】下面程序定义 q 作为 a 的引用，当接收用户输入的一个整数并将其赋值给引用 q 后，变量 a 的值也将随之变化，实现代码及结果如图 7-56 所示。

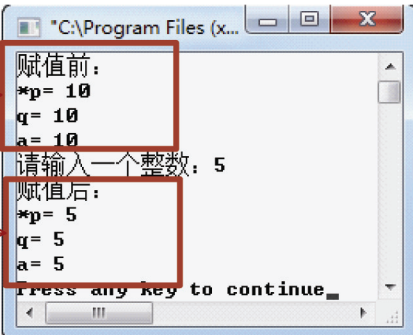
```
#include <iostream h>
int main()
{
    int a=10, b;
    int *p=&a;
    int &q=a;
    cout<<"赋值前: "<<endl;
    cout<<"*p= "<<*p<<endl;
    cout<<"q= "<<q<<endl;
    cout<<"a= "<<a<<endl;

    cout<<"请输入一个整数: ";
    cin>>b;
    p=&b;
    q=b;
    cout<<"赋值后: "<<endl;
    cout<<"*p= "<<*p<<endl;
    cout<<"q= "<<q<<endl;
    cout<<"a= "<<a<<endl;
    return 0;
}
```

定义整型指针

定义整型引用

指针、引用再次赋值



赋值前:  
\*p= 10  
q= 10  
a= 10  
请输入一个整数: 5  
赋值后:  
\*p= 5  
q= 5  
a= 5  
Press any key to continue

图 7-56 指针与引用实例

指针可以被重新赋值以指向另一个不同的对象，而引用总是指向在初始化时被指定的对象，以后不能改变。例如，在上述程序中将语句“q=b;”修改为“&q=b;”，即改变引用的值，编译器将会给出错误信息。

### 7.6.3 引用作为函数参数

对变量的引用相当于变量的别名，以“int &m=n;”为例，对 m 的操作实质上就是对 n 的操作，所以 m 既不是 n 的副本，也不是指向 n 的指针，m 就是 n 的本身。

C++允许函数通过引用实现参数传递。在调用函数时，不会再为形参分配内存空间，因为此时形参就是实参本身，在函数中对形参进行的任何操作，本质上都是对调用函数中的实参进行的操作。

【示例 7-19】下面的程序将指针作为函数的参数进行传递，完成两个数之间的互相交换，使用的是引用的传递方式。其实现代码及结果如图 7-57 所示。



```
#include<iostream>

using namespace std

int main()
{
    void change(int &,int &);
    //change() 函数声明，传递的是引用

    int x=2,y=3;

    cout<<"变量x的地址是: "<<&x<<endl;

    cout<<"交换前: x= "<<x<<", y= "<<y<<endl;
    change(x,y);

    //change() 函数的调用

    cout<<"交换后: x= "<<x<<", y= "<<y<<endl;
    return 0;
}

void change(int &n,int &m)
    //change() 函数的定义

{
    int temp;
    temp=n;
    n=m;
    m=temp;
    cout<<"形参n的地址是: "<<&n<<endl;
}
```

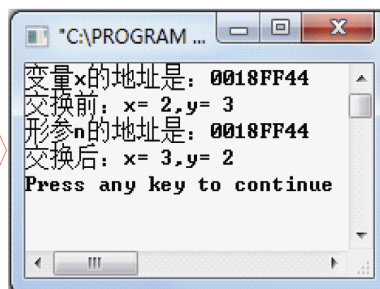


图 7-57 引用作为函数参数

代码“change(x,y);”通过 change()函数的引用传递实现了 x 和 y 数值的互换，从输出的实参 x 和形参 n 的地址可以看出，两者对应同一片内存区域。这说明在调用函数 change()时，并没有给形参 m 和 n 分配内存空间，而是直接对实参进行操作的。

## 7.7 小结

本章主要介绍了 C++中指针和引用的相关内容。对于指针，着重讲解了指针的概念、指针的访问和一些特殊指针。根据指针的应用，以具体示例介绍了指向一维数组、二维数组、字符串和函数的指针。此外，还简单介绍了指针的算术运算和动态内存分配。最后，介绍了引用及其与指针的区别。



## 7.8 习题

【题目 7-1】定义一个实型变量和一个指向实型的指针变量，并使指针变量指向定义的实型变量。要求：

- (1) 输出实型变量的内存地址。
- (2) 指针变量的内存地址。
- (3) 指针变量的值。
- (4) 用指针访问实型变量。
- (5) 指针变量所占用的内存空间大小。
- (6) 指针所指的变量占用内存的大小。

运行结果如图 7-58 所示（注意：地址值可能不一样）。

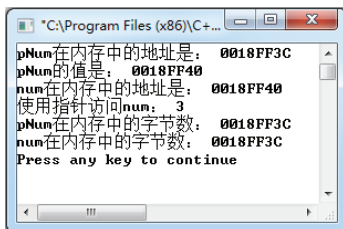


图 7-58 运行结果

【题目分析】本题考查指针变量的声明等指针的基本知识。

【关键代码】

```
double num=3;
double *pNum=&num;
cout<<"pNum 在内存中的地址是: "<<&pNum<<endl;
cout<<"pNum 的值是: "<<pNum<<endl;
cout<<"num 在内存中的地址是: "<<&num<<endl;
cout<<"使用指针访问 num: "<<*pNum<<endl;
cout<<"pNum 在内存中的字节数: "<<&pNum<<endl;
cout<<"num 在内存中的字节数: "<<&num<<endl;
```

【题目 7-2】赵红和李宁在学习了 C++语言的指针后，对指针的认识产生了分歧。赵红认为指针变量占用的内存空间和指针所指的变量有关。例如，若一个指向整型变量的指针与整型变量所占的空间一样大。而李宁认为，指针变量占用的空间应该是不变的。指针变量不会因为所指的变量占用的空间大就相应增多。请利用所学的知识，判断谁说的有道理。要求：编写程序，定义 3 个基本类型的变量，分别用不同的指针指向它们，然后测定它们所占的内存空间大小。程序的运行效果如图 7-59 所示。

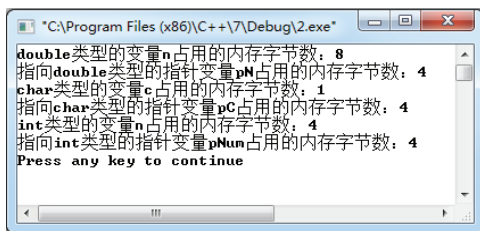


图 7-59 运行结果





【题目分析】指针变量中存放的是一个内存地址。对于不同的系统，指针变量所占用的内存空间大小不定。但是，指针变量在一个确定的系统中所占用的内存空间是一定的，并不会因为其所指的变量而变化。

【关键代码】

```
double n;
double *pN=&n;
char c;
char *pC=&c;
int num;
int *pNum=&num;
cout<<"double 类型的变量 n 占用的内存字节数: "<<sizeof(n)<<endl;
cout<<"指向 double 类型的指针变量 pN 占用的内存字节数: "<<sizeof(pN)<<endl;
cout<<"char 类型的变量 c 占用的内存字节数: "<<sizeof(c)<<endl;
cout<<"指向 char 类型的指针变量 pC 占用的内存字节数: "<<sizeof(pC)<<endl;
cout<<"int 类型的变量 n 占用的内存字节数: "<<sizeof(num)<<endl;
cout<<"指向 int 类型的指针变量 pNum 占用的内存字节数: "<<sizeof(pNum)<<endl;
```

【题目 7-3】定义一个指针变量 p，并对其进行初始化。再定义一个指针变量 q，用指针 p 对其初始化。利用指针变量 p、q 修改所指变量的值。程序的执行效果如图 7-60 所示。

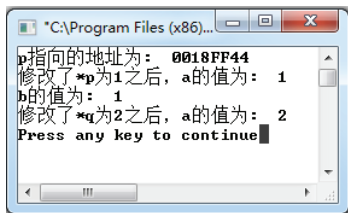


图 7-60 运行结果

【题目分析】本题主要考查指针变量的定义、指针变量的初始化，以及如何在指针和变量之间建立联系，如何使用指针访问变量。

【关键代码】

```
int a=0;
int *p=&a;
cout<<"p 指向的地址为: "<<p<<endl;
*p=1;
cout<<"修改了*p 为 1 之后, a 的值为: "<<a<<endl;
int b=*p;
cout<<"b 的值为: "<<b<<endl;
int *q=p;
*q=2;
cout<<"修改了*q 为 2 之后, a 的值为: "<<a<<endl;
```

【题目 7-4】利用指针反转数组。要求：数组中的元素从键盘输入，并且输入和输出时都使用指针进行遍历。进行反转数组操作时必须使用指针对数组元素进行操作。最后输出反转后的数组。程序的运行效果如图 7-61 所示。

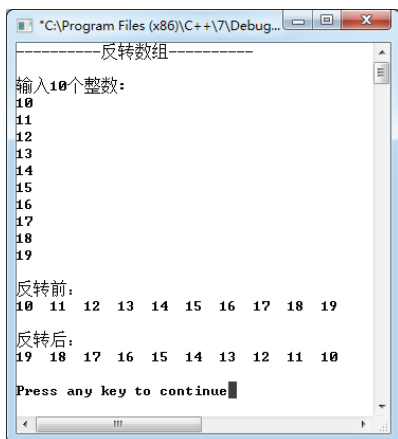


图 7-61 运行结果

【题目分析】本题考查数组指针和如何使用数组指针访问数组元素。在使用指针访问数组需要注意不要越界，即保证指针指向数组第一个到最后一个元素，超出这个范围程序可能会崩溃。

【关键代码】

```
cout<<"-----反转数组-----"<<endl;
cout<<endl;

int array[10];
int i;
int *p;
p=array;
cout<<"输入 10 个整数: "<<endl;
for(i=0;i<10;i++)
{
    cin>>*(p+i);
}
cout<<endl;
p=array;

cout<<"反转前: "<<endl;
for(i=0;i<10;i++)
{
    cout<<*(p+i)<<" ";
}
p=array;
cout<<endl<<endl;

int *q=p+((sizeof(array)/sizeof(array[0]))-1);
while(p<q)
{
    int t=*p;
    *p=*q;
    *q=t;

    p++;
    q--;
}
p=array;

cout<<"反转后: "<<endl;
```



```
for(i=0;i<10;i++)
{
    cout<<*(p+i)<<" ";
}
cout<<endl<<endl;
```

【题目 7-5】定义一个函数，交换两个变量的值。要求：形参变量值的交换要影响调用函数中变量的值。程序运行结果如图 7-62 所示。

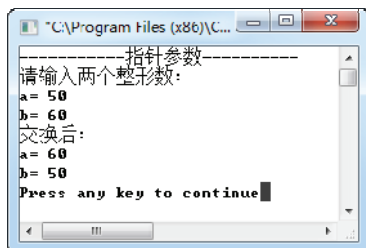


图 7-62 运行结果

【题目分析】本题主要考查指针作为函数参数的使用。注意：指针作为函数的参数时，参数的传递方式依然是值传递。因为指针作参数时，并没有改变指针本身的值，而是改变了指针所指变量的值。

#### 【关键代码】

```
int main()
{
    void swap(int *a,int *b);
    cout<<"-----指针参数-----"<<endl;
    cout<<"请输入两个整形数: "<<endl;
    int a=0,b=0;
    cout<<"a= ";
    cin>>a;
    cout<<"b= ";
    cin>>b;
    swap(&a,&b);
    cout<<"交换后: "<<endl;
    cout<<"a= "<<a<<endl;
    cout<<"b= "<<b<<endl;

    return 0;
}

void swap(int *a,int *b)
{
    int temp=*a;
    *a=*b;
    *b=temp;
}
```

【题目 7-7】动态创建一个整型变量作为数组的长度。再动态创建一个整型数组，整型数组的长度为动态创建的整型变量的大小。输入动态数组元素，再输出动态数组中的元素。程序的运行结果如图 7-63 所示。

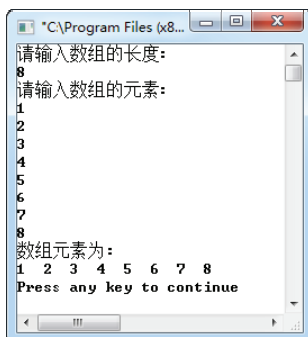


图 7-63 运行结果

【题目分析】考查使用指针进行动态内存的分配方法。

【关键代码】

```
int *pCount=new int;
cout<<"请输入数组的长度: "<<endl;
cin>>*pCount;
cout<<"请输入数组的元素: "<<endl;
int *pArray=new int(*pCount);
int i;
for(i=0;i<*pCount;i++)
{
    cin>>pArray[i];
}
cout<<"数组元素为: "<<endl;
for(i=0;i<*pCount;i++)
{
    cout<<pArray[i]<<" ";
}
cout<<endl;

delete pCount;
delete pArray;
```

【题目 7-8】编写一个程序计算 1~100 之间的连续累加，要求用指针实现。要求：第一个位置放 1，第二个位置放 1+2，第三个位置放 1+2+3，以此类推。程序的运行结果如图 7-64 所示。

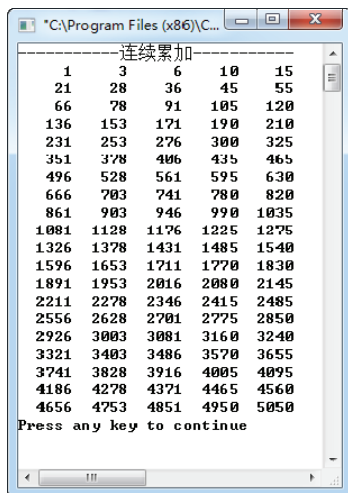


图 7-64 运行结果



【题目分析】本题要求计算从 1~100 的连续累加，从要求中可以看出，实质是前一个存储位置的值加上本存储位置的编号。因此需要申请 100 个存储单元存放累加和，在第一个存储位置放 1，后面依次累加即可。

【关键代码】

```
int *sum=NULL;
int *p;
int x=0;
sum=new int[100];
*sum=1;
p=sum;
int i=0;
for(i=2;i<=100;i++)
{
    x=*sum;
    sum++;
    *sum=x+i;
}
sum=p;
int j=0;
for(i=0;i<100;i++)
{
    cout<<setw(6)<<setfill(' ')<<*sum;
    sum++;
    j++;
    if(j==5)
    {
        cout<<endl;
        j=0;
    }
}
sum=p;

delete [] sum;
```

【题目 7-9】定义一个 5 行 5 列的整型二维数组，用指针变量访问这个二维数组。程序的运行结果如图 7-65 所示。

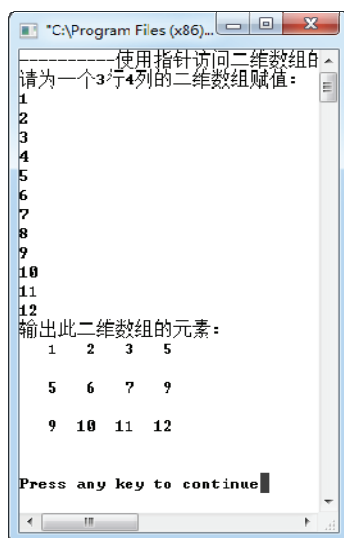


图 7-65 运行结果



【题目分析】考查使用指针访问多维数组的方法。

【关键代码】

```
int sz[3][4];
int *psz;
psz=sz[0];
cout<<"-----使用指针访问二维数组的元素-----"<<endl;
cout<<"请为一个 3 行 4 列的二维数组赋值: " <<endl;
int i;
int j;
for(i=0;i<3;i++)
{
    for(j=0;j<4;j++)
    {
        cin>>*(psz+i*3+j);
    }
}

cout<<"输出此二维数组的元素: " <<endl;
for(i=0;i<3;i++)
{
    for(j=0;j<4;j++)
    {
        cout<<setw(4)<<setfill(' ')<<*(psz+i*3+j);
    }
    cout<<endl<<endl;
}
cout<<endl;
```

【题目 7-10】使用字符串处理函数 `strcmp` 判断两个字符串是否相等。要求：调用 `strcmp` 时使用函数指针进行调用。程序的运行结果如图 7-66 所示。

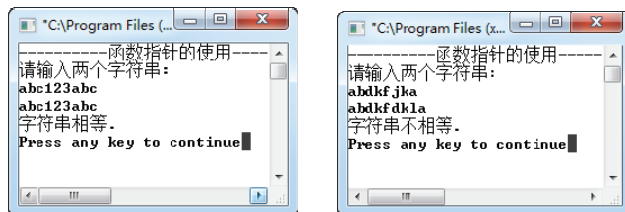


图 7-66 运行结果

【题目分析】本题考查函数指针的使用方法。

【关键代码】

```
char str1[128];
char str2[128];
cout<<"请输入两个字符串: " <<endl;
cin>>str1;
cin>>str2;
int (*pF)(const char*,const char*);
pF=strcmp;
int result=pF(str1,str2);
if(result==0)
{
    cout<<"字符串相等."<<endl;
}
else
{
    cout<<"字符串不相等."<<endl;
}
```

# 第 8 章 复合数据类型

前面的章节介绍了基本数据类型，但在具体的应用中，这些基本数据类型不能完全满足程序设计的需要，为此，C++引入了复合数据类型。本章将介绍几种常用的复合数据类型：结构体、联合、枚举和用户自定义数据类型。

## 8.1 结构体

结构体用于解决在一个数据类型中包含多个数据类型成员的问题，其在现实应用中较为常见。本节将详细介绍结构体及其定义和声明。

### 8.1.1 结构体概述

在实际应用中，一组数据往往具有不同的数据类型。例如，在学生的学籍表中，姓名应为字符型；学号可为整型或字符型；年龄应为整型；性别应为字符型；成绩可为整型或实型。这几种数据用基本数据类型表示如图 8-1 所示。

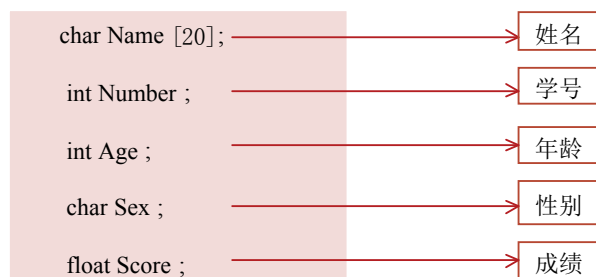


图 8-1 基本数据类型表示

对于图 8-1 中的数据，显然不能用一个数组来存放，因为数组中各元素的类型和长度都必须一致。此时就引入了一个新的复合数据类型——结构体。结构体的定义如图 8-2 所示。



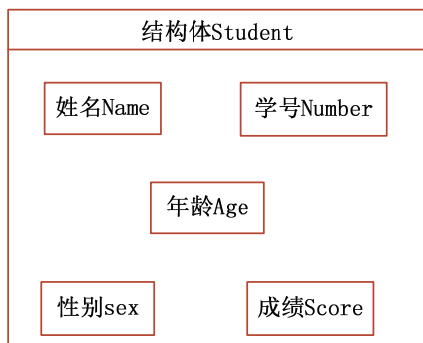
图 8-2 结构体的定义



**注意：**结构体是一种用户自定义的数据类型，因此可以自己为结构体命名。



例如，将图 8-1 所示的学生学籍表中的 5 个数据类型组成一个整体，并将其存入结构体中，为该结构体命名为 `Student` 后，这 5 个变量就成为了结构体 `Student` 的 5 个数据成员了，如图 8-3 所示。

图 8-3 结构体 `Student`

可以看出，在一个结构体 `Student` 中可以包含多个不同的数据类型，它与数组有所区别，结构体的应用比数组更为广泛。

### 8.1.2 定义结构体类型

复合数据类型与基本数据类型有所不同，如图 8-4 所示。

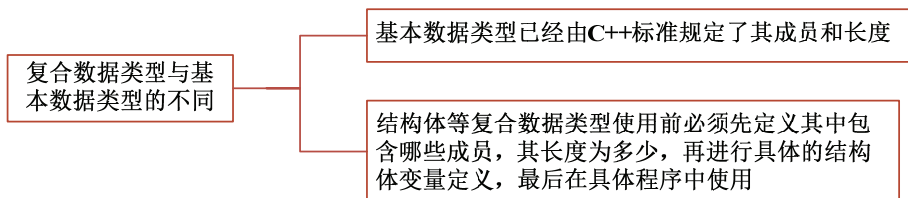


图 8-4 复合数据类型与基本数据类型的不同

在 C++ 中，用关键字 `struct` 来定义结构体，即所有结构体必须先通过 `struct` 关键字进行说明其数据成员，其一般形式如图 8-5 所示。

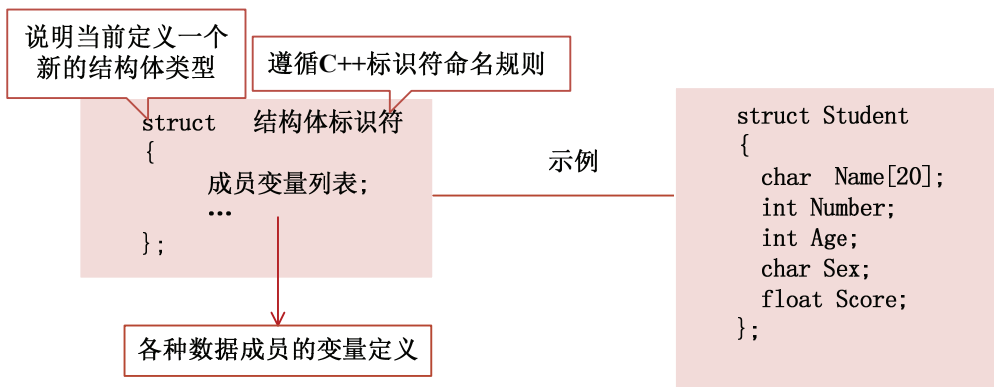


图 8-5 定义结构体



**注意：**一个结构体中的成员变量不能重名。





结构体也可以嵌套定义，即在结构体的定义中，不但可以包含基本数据类型，还可以包含已定义的结构体数据类型。例如，在定义一个学生选课表结构体时，需要使用到学生学籍表中的所有数据成员，此时就可以将已定义的 Student 结构体作为学生选课表结构体 Course 的一个成员，定义如图 8-6 所示。

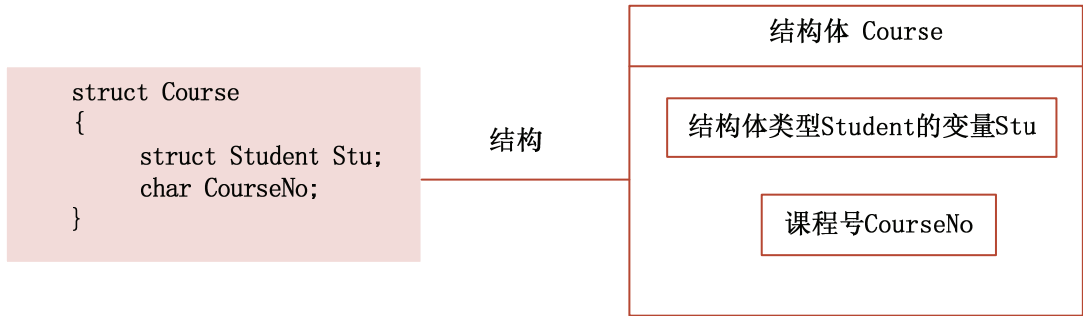


图 8-6 结构体 Course

结构体嵌套定义的优点如下：声明一个结构体时，结构体中的成员数目不宜过多，若成员数目过多结构体将不宜管理。建议将一个成员数目多的结构体进行分解。如下例所示，定义了一个员工结构体。

```
struct employeeType
{
    string firstname;
    string middlename;
    string lastname;
    string emID1;
    string address1;
    string address2;
    string citty;
    string state;
    string zip;
    int monthhire;
    int hireday;
    int hireyear;
    int quitmonth;
    int quitday;
    int quityear;
    int monthhire;
    int hireday;
    string phone;
    string cellphone;
    string fax;
    string pager;
    string email;
    string deptID;
    string salary;
};
```

重新组织如下：

```
struct nameType
{
    string first;
    string middle;
    string last;
};
```



```
struct addressType
{
    string address1;
    string address2;
    string city;
    string zip
};

struct dateType
{
    int month;
    int day;
    int year;
};

struct contactType
{
    string phone;
    string cellphone;
    string fax;
    string pager;
    string email;
};
```

结构体嵌套定义:

```
struct employeeType
{
    nameType name;
    string empID;
    addressType address;
    dateType hireDate;
    dateType quitDate;
    contactType contact;
    string deptID;
    double salary;
};
```

在 C++ 中, 结构体在定义时并不分配内存单元, 只有在定义了结构体变量后才分配存储单元。

### 8.1.3 声明结构体变量

在定义结构体类型完成后, 该结构体类型包含的成员变量、长度和类型都已经定义好了, 接下来就可以使用该类型声明具体的结构体变量了。

例如, 构造一个结构体类型 **Student**, 并声明一个该结构体类型的变量 **Stu**, 可以用如图 8-7 所示的 4 种方法来实现。



```
struct Student
{
    char Name[20];
    int Number;
    int Age;
    char Sex;
    float Score;
};
struct Student Stu;
```

1

```
struct Student
{
    char Name[20];
    int Number;
    int Age;
    char Sex;
    float Score;
}Stu;
```

2

```
struct
{
    char Name[20];
    int Number;
    int Age;
    char Sex;
    float Score;
}Stu;
```

3

```
typedef struct
{
    char Name[20];
    int Number;
    int Age;
    char Sex;
    float Score;
}Student
Student Stu;
```

4

图 8-7 声明结构体变量的 4 种方法

第 4 种方法中通过 `Student Stu` 即可完成结构体变量 `Stu` 的声明。此处的 `Student` 是一个具体的结构体类型名，它能够唯一地标识这种结构体类型。因此，可用它来定义变量，如同使用 `int`、`char` 一样，不需再写关键字 `struct`。有关 `typedef` 关键字的说明在后续章节中将具体介绍。



## 8.2 结构体的应用

定义结构体类型和声明结构体变量后，就可以在具体程序中使用该结构体了。一般来说，在使用前可以为结构体变量赋初值，以便在程序中引用结构体成员变量。

### 8.2.1 初始化结构体变量

结构体变量的初始化方式与数组类似，分别给结构体的成员变量以初始值。而结构体成员变量的初始化遵循简单变量或数组的初始化方法。

根据 8.1.3 节声明结构体变量的不同方法，初始化变量的形式也有多种。例如，下列语句对上述结构体类型 `struct Student` 声明的变量 `Stu` 进行初始化，如图 8-8 所示。



```
struct Student
{
    char Name [20];
    int Number ;
    int Age ;
    char Sex ;
    float Score ;
};
struct Student Stu ={"张三", 20101, 21, 'M', 95 };
```

对应

结构体变量Stu				
Name	Number	Age	Sex	Score
张三	20101	21	M	95

图 8-8 初始化结构体变量



**注意：**对结构体变量进行赋初值时，C++编译程序按每个成员在结构体中的顺序一一对应赋初值，不允许跳过前面的成员给后面的成员赋初值。

同样，其他几种在定义结构体类型的同时声明变量的初始化方法类似，只需直接将初始化列表写入结构体变量后，用一对“{}”包围起来即可。

此外，C++允许只给前面的若干成员赋初值，对于后面未赋初值的成员，系统自动赋默认的值。例如，如果将上述初始化语句改写为：“struct Student Stu={"张三"};”，那么该结构体变量 Stu 中各成员变量对应的值如图 8-9 所示。

结构体变量Stu				
Name	Number	Age	Sex	Score
张三	0	0	\0x0	0.0

int类型的变量默认值为 0，  
char类型的变量默认值为空（'0x0'），  
float类型的变量默认值为 0.0

图 8-9 默认初值



相同类型的结构体变量之间可以进行整体赋值。例如，mybox 和 yourbox 都是 box 类型结构体定义的变量，它们之间可以进行整体赋值。

```
struct box{
    int length;           //长
    int width;            //宽
    int height;           //高
};
struct box mybox,yourbox;
mybox.length=10;
mybox.width=19;
mybox.height=15;
yourbox=mybox;
```

其中，“yourbox=mybox;”等价于下面的语句：

```
yourbox.length=mybox.length;
yourbox.width=mybox.width;
yourbox.height=mybox.height;
```

## 8.2.2 引用结构体变量成员

由于结构体变量由多个不同类型的成员变量组成，因此在具体程序中需引用其中的成员变量。C++中引用结构体变量中的成员的基本形式如图 8-10 所示。

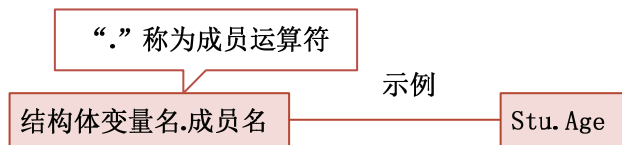


图 8-10 引用结构体变量的基本形式

## 8.2.3 结构体指针

结构体指针是指声明一个指针变量来指向一个结构体数据类型。在 C++中，指向结构体变量的指针的一般定义形式如图 8-11 所示。

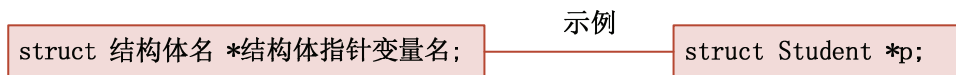


图 8-11 结构体指针定义的一般形式

常用指针形式来引用结构体中的成员变量，其形式如图 8-12 所示。

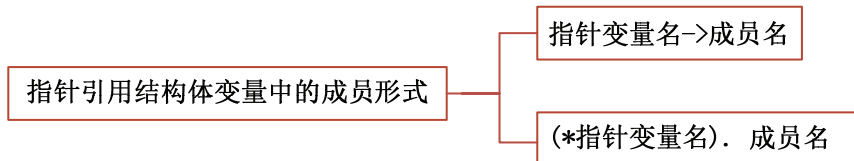


图 8-12 指针形式引用结构体中的成员变量形式

【示例 8-1】下面的程序定义了一个结构体类型 Student，以及其变量 Stu 和指向结构体 Student 的指针 p，在主程序中通过 3 种方式引用该结构体的成员变量，实现代码及结果如图 8-13 所示。



图 8-13 3 种方式引用结构体的成员变量

## 8.2.4 结构体数组

除了变量可以是结构体类型外，数组的元素也可以是结构体类型，因此可以构成结构型数组。结构体数组的每一个元素都是具有相同结构类型的结构变量。

在实际应用中，经常用结构体数组来表示具有相同数据结构的一个群体。例如，学生学籍表中需要保存 50 个学生的姓名、学号、年龄、性别和成绩等信息，而一个结构体变量只能保存一个学生的这些信息，此时就可以通过定义结构体数组来实现。结构体数组的定义与结构体变量的声明类似，实现代码如图 8-14 所示。

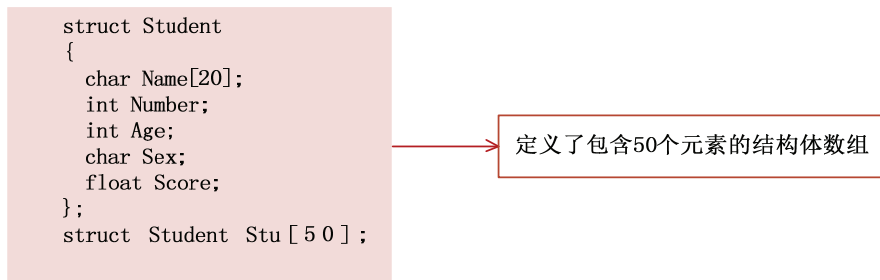


图 8-14 定义结构体数组



此外, 结构体数组的赋初值可以在定义数组的同时进行, 每个数组元素都包含结构体的所有成员变量。

【示例 8-2】下面的程序对一个包含两个元素的结构体数组进行初始化后将其输出, 实现代码及结果如图 8-15 所示。

```
#include <iostream h>
struct Student
{
    char Name[20];
    int Number;
    int Age;
    char Sex;
    float Score;
};
int main()
{
    struct Student Stu[2]={ { "张三", 20101, 21, 'M', 95 },
                             { "李四", 20103, 23, 'F', 100 } };

    cout<<Stu[i].Name<<" "<<Stu[i].Number<<" "<<Stu[i].Age<<" "<<Stu[i].Sex
    <<" "<<Stu[i].Score<<" "<<endl;

    cout<<endl;
    return 0;
}
```

定义了包含两个元素的结构体数组, 并初始化

cout<<Stu[i].Name<<" "<<Stu[i].Number<<" "<<Stu[i].Age<<" "<<Stu[i].Sex <<" "<<Stu[i].Score<<" "<<endl;

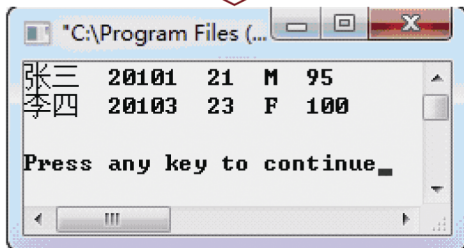


图 8-15 结构体数组实例



**说明:** 结构体类型的定义和结构体变量、数组的声明既可以放在主函数 main() 中, 也可以放在外面。

由于结构体变量由多个不同类型的成员变量组成, 因此在具体程序中需引用其中的成员变量。C++ 中引用结构体变量中的成员的基本形式如图 8-16 所示。

“.” 称为成员运算符

示例

结构体变量名.成员名

Stu. Age

图 8-16 结构体变量成员的引用



### 8.2.5 结构体和数组的比较

由于结构体和数组都是复合类型，它们有很多相似的地方，也有各自的特点。表 8-1 对二者进行了详细的比较。

表 8-1 结构体和数组

操作	数组	结构体
算术运算	否	否
赋值运算	否	是
输入/输出	否（除了字符数组）	否
比较运算	否	否
参数传递	只能引用传递	值传递和引用传递
函数返回值	否	是

## 8.3 联合

联合也称共用体（union），可以看成一种特殊的结构。与结构体一样，联合可以包括多个数据类型。但在联合中，各种数据类型在内存中占据同一地址。本节将详细讲解联合的声明定义与引用。

### 8.3.1 定义联合类型

联合与结构体有一些相似之处，但两者有本质上的不同，如图 8-17 所示。

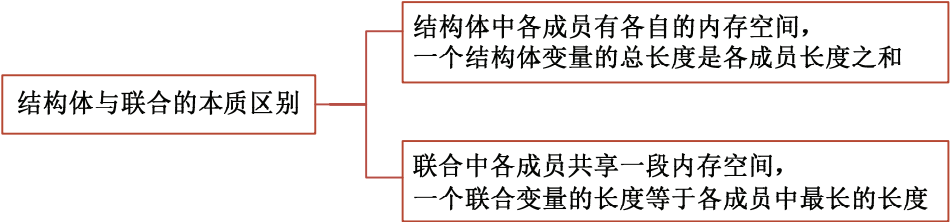


图 8-17 结构体与联合的本质区别

在 C++ 中，联合数据类型的声明与结构体基本相同，区别在于其声明关键字为 union。定义一个联合数据类型的语法形式如图 8-18 所示。

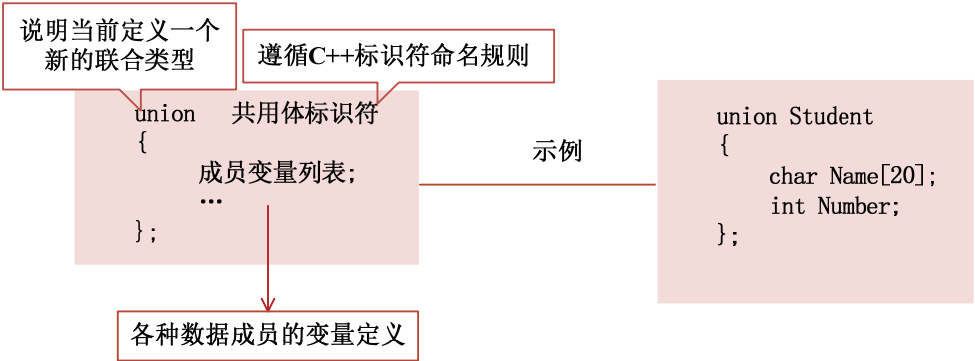


图 8-18 定义联合类型





### 8.3.2 声明联合变量

联合数据类型定义完成之后，就可以在具体程序中声明该类型的变量。与结构体数据类型变量的声明类似，联合变量的声明也有4种方式，如图8-19所示。

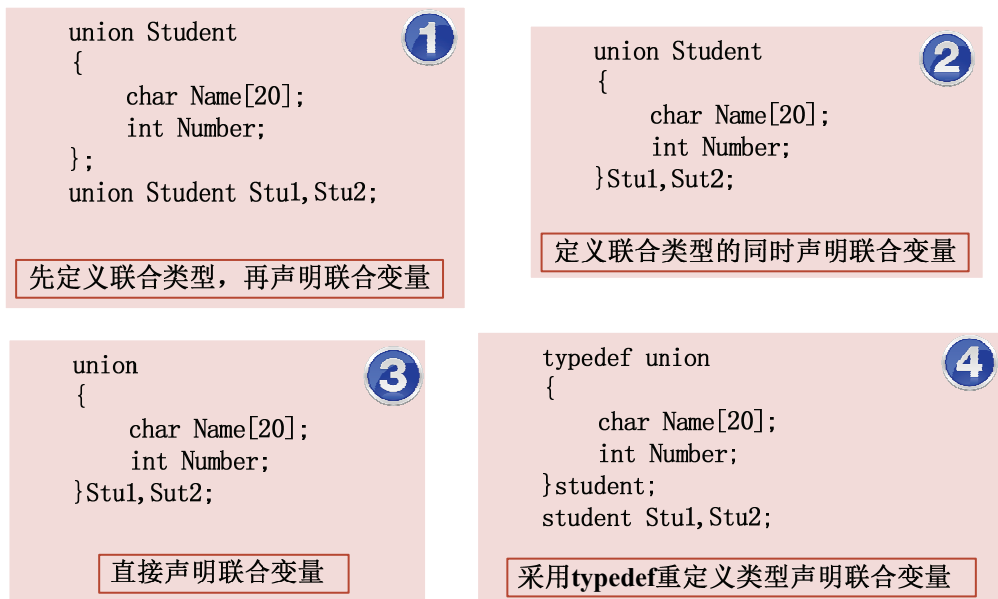


图 8-19 联合声明变量的4种方式



**注意：**上述4种方法声明的联合变量Stu1和Stu2都是完全相同的，在存储空间上，其成员变量共享一个内存空间。

### 8.3.3 引用联合类型成员

与结构体成员变量的引用类似，定义了联合变量和指向该联合的指针变量后，联合变量中每个成员的引用有3种形式，如图8-20所示。

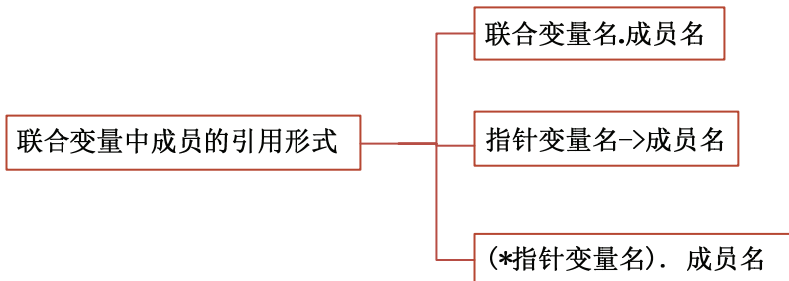


图 8-20 联合变量中每个成员的引用形式

**【示例 8-3】**下面的程序定义了一个联合数据类型，并声明了该类型的变量和指针，接收用户从键盘的输入后将变量的成员变量值输出，其实现代码及结果如图8-21所示。



```
#include <iostream.h>
```

```
union Student
```

```
{
```

```
    char Name[20];
```

```
    int Number;
```

```
    int Age;
```

```
    char Sex;
```

```
    float Score;
```

```
};
```

```
int main()
```

```
{
```

```
    union Student Stu;*p;
```

```
    cout<<"请输入联合变量的姓名值 : "<<endl;
```

```
    cin>>Stu.Name;
```

```
    p=&Stu;——> 结构体指针初始化
```

```
    cout<<"Stu.Name= "<<Stu.Name<<endl;
```

```
    cout<<"Stu.Number= "<<Stu.Number<<endl<<endl;
```

```
    cout<<"(*p).Name= "<<(*p).Name<<endl;
```

```
    cout<<"(*p).Number= "<<(*p).Number<<endl<<endl;
```

```
    cout<<"p->Name= "<<p->Name<<endl;
```

```
    cout<<"p->Number= "<<p->Number<<endl;
```

```
    return 0;
```

```
}
```

定义了包含5个成员变量的联合类型

定义联合变量和指针

结构体指针初始化

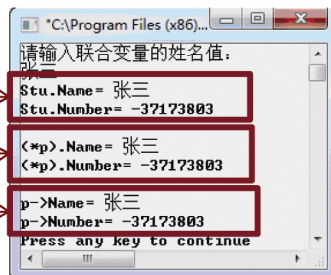


图 8-21 3 种方式引用联合变量的成员

从如图 8-20 所示的结果图中可以看出，程序接收用户输入的“张三”后，将其存储到联合变量 Stu 的第一个成员 Name 中，而其他成员变量则没有值，因此输出 Number 成员的值是一个无意义的值，这是由联合数据类型的特征决定的。



**注意：**联合体类型的变量中存储的内容是最后被写进去的值。



## 8.4 枚举

枚举数据类型也是 C++ 中常见的一种复合数据类型。枚举将现实生活中表示同一种类型的各个常量融合为一个整体，以便在具体程序中统一调用。本节将详细介绍枚举类型的定义、声明和引用。

### 8.4.1 定义枚举类型

在日常生活中，会遇到很多集合类问题，其所描述的状态为有限的几个。例如，考试的结果有优秀、良好、中等、及格和不及格 5 种；天气有晴、多云、阴、雨等几种状态。在计算机中表述这些信息，需要定义一组整型常量，但是这些常量虽然表达了同一类型的信息，但是在



语法上是彼此孤立的个体，而不是一个完整的逻辑整体。

枚举类型的引入解决了将孤立的常量融合为一个逻辑整体的问题。例如，将考试的结果用枚举类型来实现，该类型就包含 5 个常量，如图 8-22 所示。

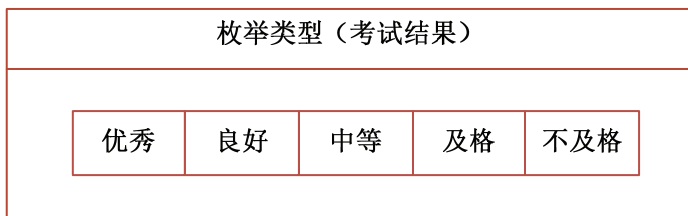


图 8-22 枚举类型

在 C++ 中，枚举数据类型的定义与结构体和联合类似，其一般形式如图 8-23 所示。

说明当前定义一个  
新的枚举类型

enum 枚举标识符 {常量列表};

示例

enum exam {you, liang, zhong, jige, bujige};

包含该枚举类型的值，每个枚举常量之间用逗号分隔

图 8-23 定义枚举数据类型

此外，还可以为枚举常量指定其对应的整型常量数值，如为枚举常量 you 指定值 1，为枚举常量 liang 指定为 2 等。如果在给定枚举常量时不指定其对应的整数常量值，系统将自动为每一个枚举常量设定对应的整数常量值，如图 8-24 所示。

```
#include <iostreamh>
enum exam{you, liang, zhong, jige, bujige};
int main()
{
    cout<<"枚举变量的值："<<endl;
    cout<<"you= " <<you<<endl;
    cout<<"liang= " <<liang<<endl;
    cout<<"zhong= " <<zhong<<endl;
    cout<<"jige= " <<jige<<endl;
    cout<<"bujige= " <<bujige<<endl;
    return 0;
}
```

定义枚举类型

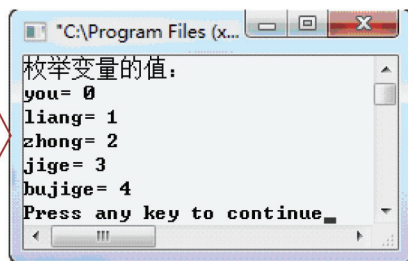


图 8-24 系统自动为枚举常量设定整型常量值

在图 8-23 所示的结果图中，C++ 编译器自动为其指定从 0 开始的值。而当用户指定其中的一个枚举常量值后，其后的所有常量都在前一个常量值基础上加 1，如图 8-25 所示。

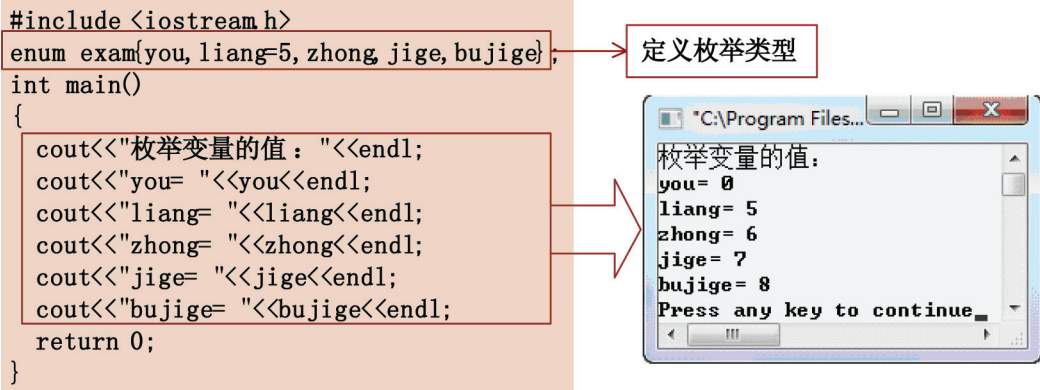


图 8-25 用户指定其中的一个枚举常量值

### 8.4.2 声明枚举变量

定义了枚举类型后，就可以接着声明枚举类型的变量了。与结构体变量和联合变量的定义类似，枚举类型的变量声明需要先定义枚举类型，一般有两种方式，如图 8-26 所示。

```
enum exam{you, liang, zhong, jige, bujige};
enum exam Stu1, Stu2;
```

1

先定义枚举类型，再声明枚举变量

```
enum exam{you, liang, zhong, jige, bujige}Stu1, Stu2;
```

2

定义枚举类型的同时声明枚举变量

图 8-26 声明枚举变量的两种方式

### 8.4.3 引用枚举变量成员

在具体程序中，引用枚举变量可将其指定为某个枚举常量。事实上，枚举类型的实质是整数集合。因此，枚举变量成员的引用类似于整型类型变量的引用，并可以与整数类型的数据之间进行类型转换。

【示例 8-4】下面的程序声明几个枚举变量和指针，并为其赋初值，实现枚举变量成员值之间的算术运算，其实现代码及结果如图 8-27 所示。

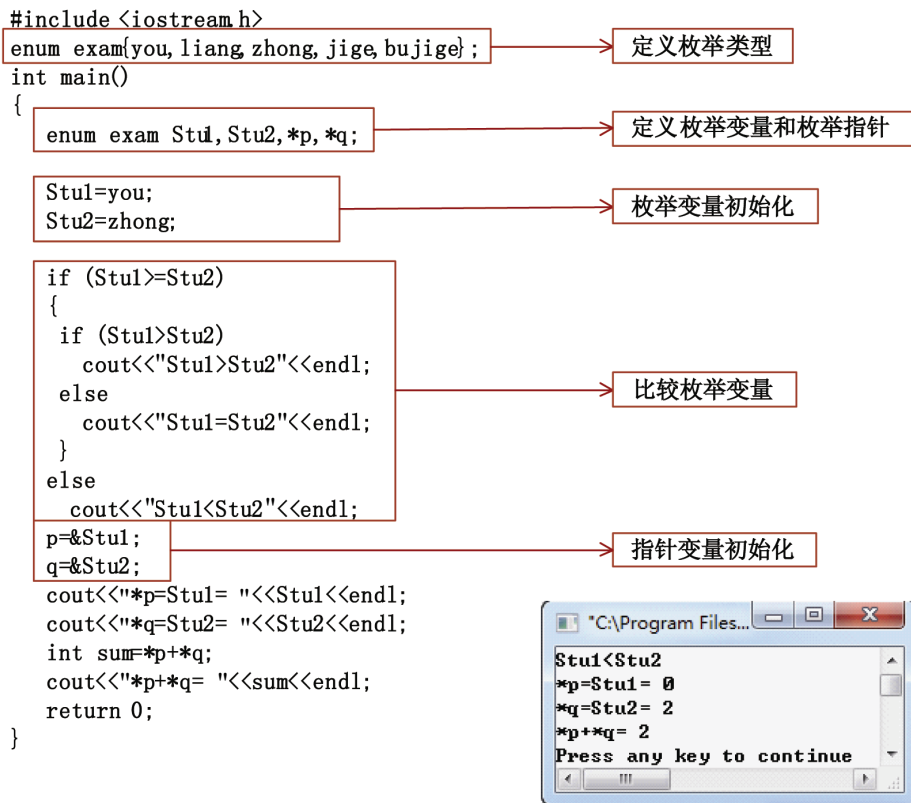


图 8-27 枚举变量实例

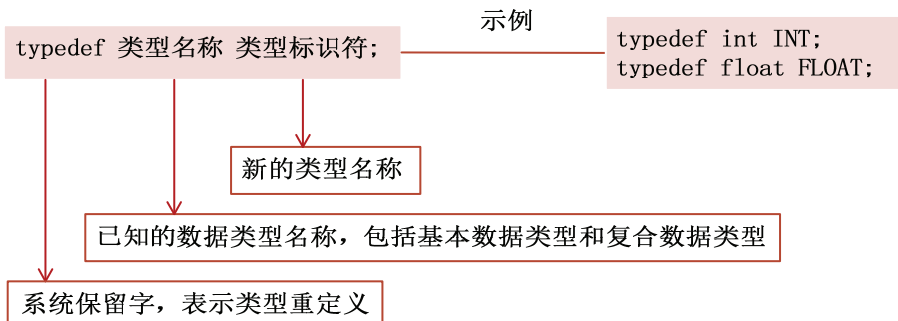


## 8.5 用户自定义数据类型

用户自定义数据类型是指 C++ 允许用户定义符合具体要求的数据类型，包括前面讲解的基本数据类型，都可以进行类型重定义，这是体现 C++ 灵活性的一个地方。

为了解决用户自定义数据类型名称的需求，C++ 引入类型重定义语句 `typedef`，其可以为数据类型定义新的类型名称，从而丰富数据类型所包含的属性信息。

在 C++ 中，类型重定义 `typedef` 的一般语法形式如图 8-28 所示。

图 8-28 类型重定义 `typedef` 的一般语法形式

【示例 8-5】下面的程序使用类型重定义 `typedef` 定义了多个类型。其在实际程序中与基本



数据类型和复合数据类型的使用方式相同，实现代码及结果如图 8-29 所示。

```
#include <iostream.h>
```

```
typedef int INT;
```

```
typedef float FLOAT;
```

```
typedef struct
```

```
{
```

```
    char Name[20];
```

```
    int Number;
```

```
    int Age;
```

```
    char Sex;
```

```
    float Score;
```

```
}Student;
```

```
int main()
```

```
{
```

```
    INT a=10;
```

```
    FLOAT b=10.01;
```

```
    cout<<"整数a= "<<a<<endl;
```

```
    cout<<"浮点数b= "<<b<<endl;
```

```
    Student Stu={"张三", 20101, 21, 'M', 95};
```

```
    Student *p;
```

```
    p=&Stu;
```

```
    cout<<endl;
```

```
    cout<<"Stu. Name= "<<Stu.Name<<endl;
```

```
    cout<<"p->Number= "<<p->Number<<endl;
```

```
    cout<<"Stu. Age= "<<Stu.Age<<endl;
```

```
    return 0;
```

```
}
```

用户自定义基本数据类型和复合数据类型

定义INT、FLOAT变量并初始化

定义结构体变量并初始化

定义结构体指针并初始化

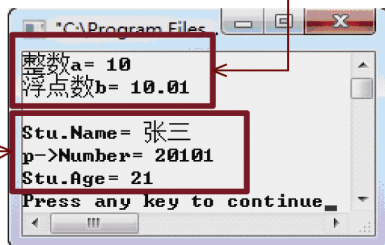


图 8-29 用户自定义数据类型实例



**注意：**在使用 typedef 语句时，并没有引入新的数据类型，而只是重新为一个已有的数据类型命名。



## 8.6 综合实例

本节将列举一些典型的例子，讲解如何运用结构体来定义和使用复杂的数据。

**【示例 8-6】**三角形的类型判断和面积计算。三角形是比较简单的二维几何图形，具有边长、角度等属性。在程序中可以利用一个结构体来表示三角形。在有关的图形计算中，常常需要对三角形进行计算和处理，这些处理可以转变成针对三角形结构体的处理。

下面的程序要求用户输入三角形的 3 条边长，判断三角形的类型，并求出三角形的面积。示例代码如图 8-30 所示，运行结果如图 8-31 所示。

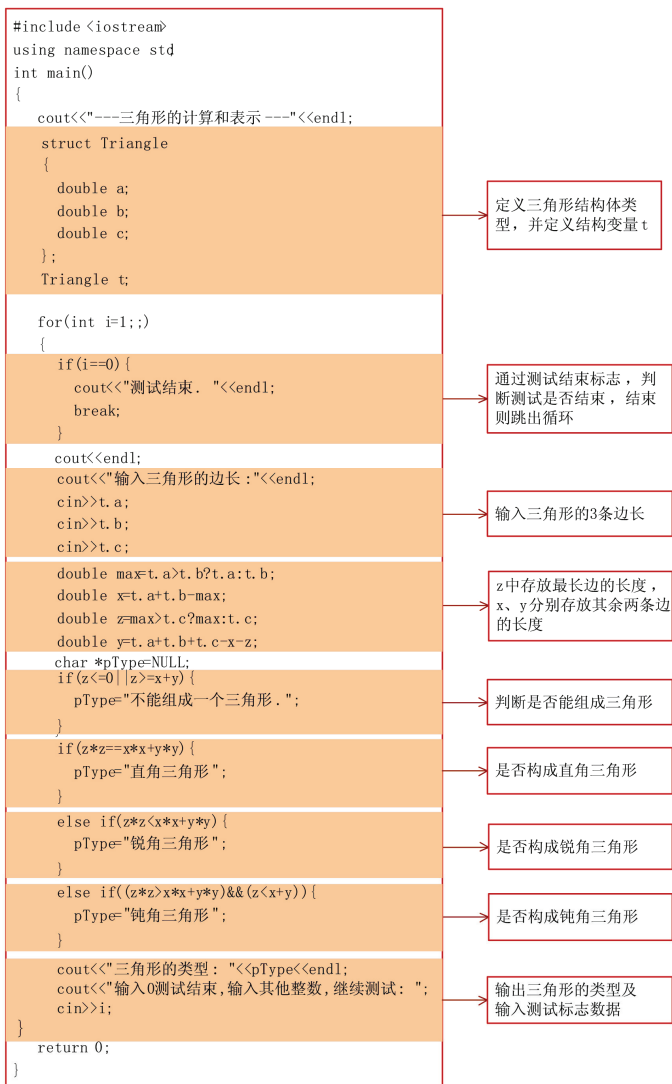


图 8-30 三角形类型判断示例代码

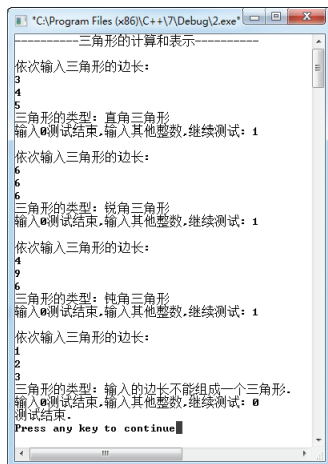


图 8-31 运行结果



【示例 8-7】下面的程序创建了一个链表，用来存储一个班级的学生数据。程序查找了学生 Jack 的信息，然后遍历输出了整个链表中所有学生的信息。程序代码如图 8-32 所示，程序的运行效果如图 8-33 所示。

<pre>#include &lt;iostream&gt; using namespace std;  struct student {     char name[20];     int num;     char sex;     int age;     student *next; };  int main() {     student stu5={"Anny", 1004, 'W', 16, NULL};     student stu4={"Jacson", 1003, 'M', 15, &amp;stu5};     student stu3={"John", 1002, 'M', 16, &amp;stu4};     student stu2={"Jack", 1001, 'M', 14, &amp;stu3};     student stu1={"Terry", 1000, 'W', 15, &amp;stu2};     student *head=&amp;stu1;     student *pointer=head;      bool isFind=false;      for(;pointer;pointer=(*pointer).next)     {         if(strcmp("Jack", (*pointer).name)==0) {             cout&lt;&lt;"要查找的学生Jack的信息为:  "&lt;&lt;endl;             cout&lt;&lt;"Jack的学号:"&lt;&lt;(*pointer).num&lt;&lt;endl;             cout&lt;&lt;"Jack的性别:"&lt;&lt;(*pointer).sex&lt;&lt;endl;             cout&lt;&lt;"Jack的年龄:"&lt;&lt;(*pointer).age&lt;&lt;endl;             isFind=true;         }     }      if(!isFind) {         cout&lt;&lt;"没有Jack这个学生. "&lt;&lt;endl;     }      pointer=head;     cout&lt;&lt;"输出链表中所有学生的信息: "&lt;&lt;endl;      for(;pointer;pointer=(*pointer).next)     {         cout&lt;&lt;(*pointer).name&lt;&lt;"的学号:"&lt;&lt;(*pointer).num&lt;&lt;endl;         cout&lt;&lt;(*pointer).name&lt;&lt;"的性别:"&lt;&lt;(*pointer).sex&lt;&lt;endl;         cout&lt;&lt;(*pointer).name&lt;&lt;"的年龄:"&lt;&lt;(*pointer).age&lt;&lt;endl;     }      return 0; }</pre>	<p>定义结构student，其中，next为下一个元素的指针</p>
	<p>用5个学生的信息初始化列表。其中，stu1是头节点，head是头指针，存放头节点stu1的地址，stu5是末尾节点</p>
	<p>用for循环遍历整个链表，查找学生，并输出学生的信息，当找到或pointer为NULL，即遍历完整个链表时结束循环。找到后将布尔变量置为 true</p>
	<p>若没有找到，输出提示信息</p>
	<p>通过循环遍历整个链表，将链表中所有的学生信息输出</p>

图 8-32 链表示例



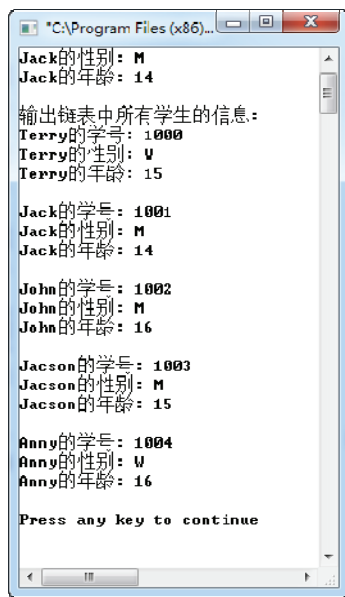


图 8-33 运行效果

## 8.7 小结

本章重点讲解了 C++ 中结构体、联合和枚、用户自定义数据类型等几种复合数据类型的声明定义和引用。并且通过示例详细讲解了其使用。此外，还讲解了常用的 typedef 重定义语句。

## 8.8 习题

【题目 8-1】从命令行读入数据，输出对应的星期名称。要求：将星期定义为枚举变量，从命令行读入一个 0~7 之间的数字。0 表示退出，1~7 分别对应输出星期名称，以及在枚举变量中的编号。程序运行结果如图 8-34 所示。

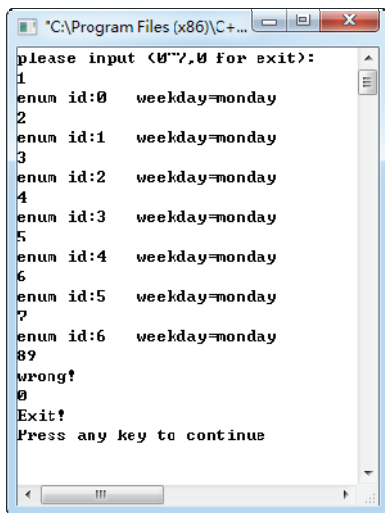


图 8-34 运行结果



【题目分析】考查枚举类型的基本知识。枚举类型的变量取值不能超过定义的范围。枚举类型值的编号是定义枚举类型时自动获得的一个整数值，该值从 0 开始。

【关键代码】

```
enum weekday {monday,tuesday,wednesday,thursday,friday,saturday,sunday} w;  
int i;  
cout<<"please input (0~7,0 for exit): "<<endl;  
do{  
    cin>>i;  
    switch(i)  
    {  
        case 1:  
            w=monday;  
            cout<<"enum id:"<<w<<"    weekday="<<"monday"<<endl;  
            break;  
        case 2:  
            w=tuesday;  
            cout<<"enum id:"<<w<<"    weekday="<<"monday"<<endl;  
            break;  
        case 3:  
            w=wednesday;  
            cout<<"enum id:"<<w<<"    weekday="<<"monday"<<endl;  
            break;  
        case 4:  
            w=thursday;  
            cout<<"enum id:"<<w<<"    weekday="<<"monday"<<endl;  
            break;  
        case 5:  
            w=friday;  
            cout<<"enum id:"<<w<<"    weekday="<<"monday"<<endl;  
            break;  
        case 6:  
            w=saturday;  
            cout<<"enum id:"<<w<<"    weekday="<<"monday"<<endl;  
            break;  
        case 7:  
            w=sunday;  
            cout<<"enum id:"<<w<<"    weekday="<<"monday"<<endl;  
            break;  
        case 0:  
            cout<<"Exit!"<<endl;  
            break;  
        default:  
            cout<<"wrong!"<<endl;  
    }  
}while(i!=0);
```

【题目 8-2】编写程序代码，定义一个学生信息的结构，并对其进行初始化，最后输出结果。程序的运行结果如图 8-35 所示。

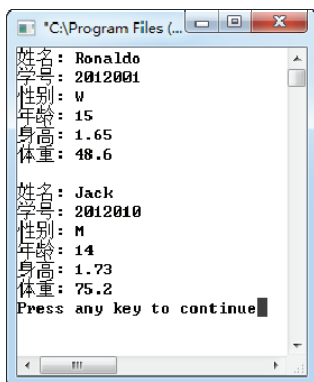


图 8-35 运行结果

【题目分析】主要考查结构体的定义及使用知识。重点是掌握如何声明、初始化和使用。

【关键代码】

```
struct student
{
    char name[20];
    int number;
    char sex;
    int age;
    double high;
    double weight;
};
student stu1={"Ronaldo",2012001,'W',15,1.65,48.6};
student stu2={"Jack",2012010,'M',14,1.73,75.2};
cout<<"姓名: "<<stu1.name<<endl;
cout<<"学号: "<<stu1.number<<endl;
cout<<"性别: "<<stu1.sex<<endl;
cout<<"年龄: "<<stu1.age<<endl;
cout<<"身高: "<<stu1.high<<endl;
cout<<"体重: "<<stu1.weight<<endl<<endl;
```

【题目 8-3】定义联合体 info，可以保存学生的年级或者老师所在的学校部门，然后根据用户输入的数据进行相应的输出。程序运行效果如图 8-36 所示。

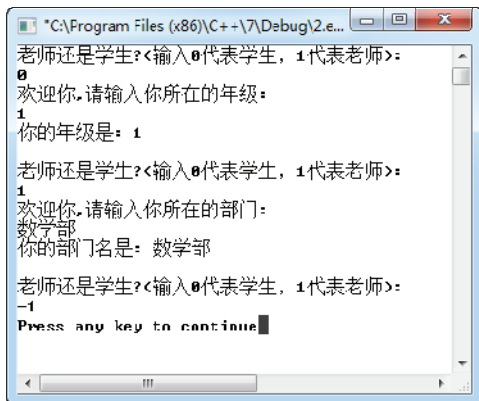


图 8-36 运行结果

【题目分析】本题主要考查联合体的相关知识，重点是掌握联合体的特点及其使用过程。

**【关键代码】**

```
union info
{
    int grade;
    char department[20];
};
info personInfo;
int ans=0;
for(;ans!=-1;)
{
    cout<<"老师还是学生?(输入 0 代表学生, 1 代表老师): "<<endl;
    cin>>ans;
    if (ans==0)
    {
        cout<<"欢迎你, 请输入你所在的年级: "<<endl;
        cin>>personInfo.grade;
        cout<<"你的年级是: "<<personInfo.grade<<endl<<endl;
    }

    if (ans==1)
    {
        cout<<"欢迎你, 请输入你所在的部门: "<<endl;
        cin>>personInfo.department;
        cout<<"你的部门名是: "<<personInfo.department<<endl<<endl;
    }
}
```

**【题目 8-4】**定义一个学生信息的结构, 结构的成员有学生姓名、学号、性别、年龄。建立一个学生的结构体数组, 数组长度为 5。要求: 学生的信息要从键盘输入, 保存到建立的结构体数组中。再将数组中的内容输出。程序的运行效果如图 8-37 所示。

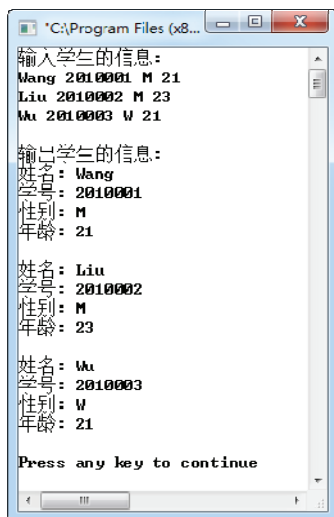


图 8-37 运行结果

**【题目分析】** 本题考查结构体数组的定义、结构体数组元素的访问方法。

**【关键代码】**

```
struct student
{
    char name[20];
    int number;
```



```

        char sex;
        int age;
    };

    student stu[3];
    int i;
    cout<<"输入学生的信息: "<<endl;
    for(i=0;i<3;i++)
    {
        cin>>stu[i].name;
        cin>>stu[i].number;
        cin>>stu[i].sex;
        cin>>stu[i].age;
    }
    cout<<endl;

    cout<<"输出学生的信息: "<<endl;
    for(i=0;i<3;i++)
    {
        cout<<"姓名: "<<stu[i].name<<endl;
        cout<<"学号: "<<stu[i].number<<endl;
        cout<<"性别: "<<stu[i].sex<<endl;
        cout<<"年龄: "<<stu[i].age<<endl;
        cout<<endl;
    }
}

```

**【题目 8-5】**同【题目 8-4】一样，但结构体成员变量的访问改为指针方式。

**【题目分析】**本题主要考查使用结构体指针访问结构变量的方法。

**【关键代码】**

```

struct student
{
    char name[20];
    int number;
    char sex;
    int age;
};

student stu[3];
student *pstu;
pstu=stu;
int i;
cout<<"输入学生的信息: "<<endl;
for(i=0;i<3;i++)
{
    cin>>pstu->name;
    cin>>pstu->number;
    cin>>pstu->sex;
    cin>>pstu->age;
    pstu=pstu+i;
}
cout<<endl;

cout<<"输出学生的信息: "<<endl;
pstu=stu;
for(i=0;i<3;i++)
{
    cout<<"姓名: "<<pstu->name<<endl;
    cout<<"学号: "<<pstu->number<<endl;
    cout<<"性别: "<<pstu->sex<<endl;
    cout<<"年龄: "<<pstu->age<<endl;
}

```



```
        pstu=pstu+i;
        cout<<endl;
    }
```

【题目 8-6】同学甲定义了一个空间点的结构体类型，如下：

```
struct tagPoint
{
    int x;
    int y;
    int z;
};
```

可他觉得每次定义点类型的结构变量时都要用 `tagPoint`，比较麻烦。请利用自定义数据类型的知识帮助他解决问题。

【题目分析】本题考查自定义数据类型的使用。

【关键代码】

```
typedef struct tagPoint
{
    int x;
    int y;
    int z;
}P;
P p1,p2;    //定义坐标类型变量
```

## 第 3 篇 C++面向对象篇

# 第 9 章 类和对象

类和对象是面向对象程序设计中最基础、最重要的两个概念。在面向对象程序设计中，所有操作都是以对象为基础的。本章将详细讲解类和对象的有关知识。

## 9.1 类和对象概述

程序中的对象是对现实对象的抽象。现实中的对象包括可感知的物体，以及思维中的概念。例如，汽车可以被看做是一个对象；学校作为一个概念也可以被看做是一个对象。现实对象有一个特点，如图 9-1 所示。

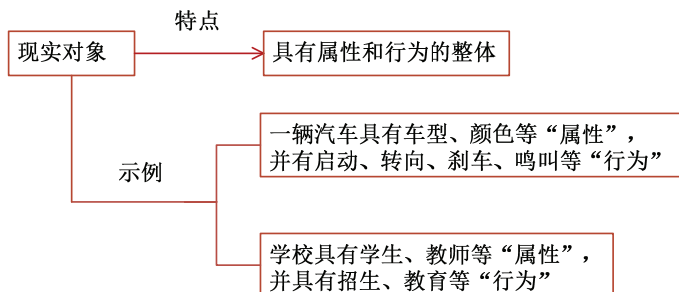


图 9-1 现实对象的特点

在程序中，属性可以抽象成数据，行为可以抽象成函数。因此对象的构成如图 9-2 所示。



图 9-2 一个对象的构成

无论是在现实世界，还是在程序中，对象都是一个完整的个体。而这些个体都可以按照某种规则进行分类。例如，一辆汽车就属于汽车类，一所学校就属于学校类。分类的规则如图 9-3 所示。

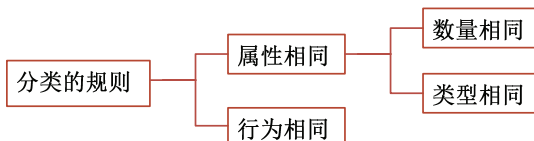


图 9-3 分类的规则





例如，所有汽车都具有车型、颜色等属性。这些属性的个数相同，不同汽车的相同属性的类型也相同。另外，所有汽车都具有启动、转向、刹车、鸣叫等行为。因此，所有汽车都可以归为汽车类，如图 9-4 所示。

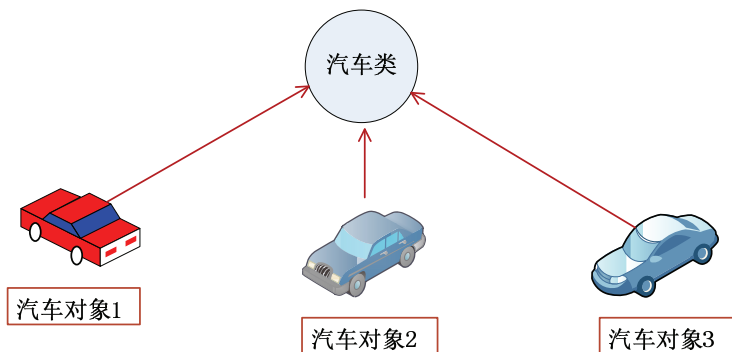


图 9-4 汽车类与汽车对象

从图 9-4 中可以看出，类是指整个一类事物，一个类定义了一个模板。类对数据处理数据的方法（函数）进行了封装，是对某一类具有相同特性和行为的事物的描述。

类和对象的关系如图 9-5 所示。

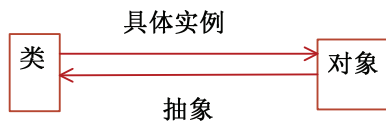


图 9-5 类和对象的关系



## 9.2 类和对象的基础语法

类（class），是将一组对象的数据结构和操作中相同的部分抽出来组成的集合，是对象共同的特征。本节简单介绍类的声明和实例化对象。

### 9.2.1 类的声明

在 C++ 中，使用关键字 `class` 来声明类，如图 9-6 所示。

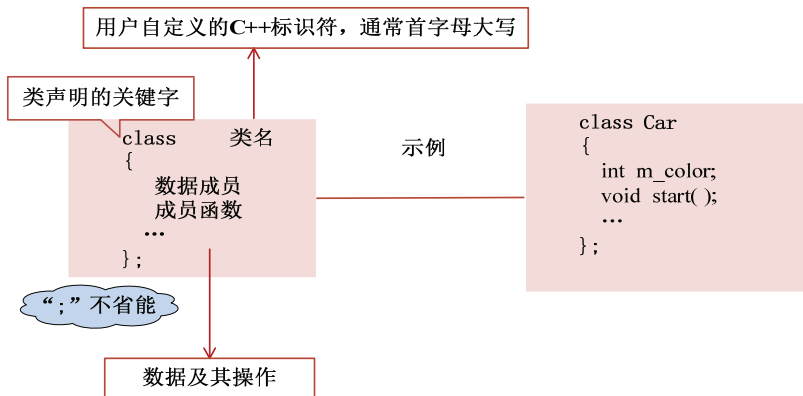


图 9-6 类的声明



在 C++ 中, `class` 是一个保留字, 它只是定义了一种数据类型, 并不分配内存单元。在类定义时, 右侧的大括号后面的分号不可以省略, 这是经常犯的错误。

`Private`、`protected`、和 `public` 是 C++ 中的保留字, 称为成员访问指示符。以下是关于三者的说明。

(1) `public` (公有成员, 包括类的属性和方法): 提供类的外部界面, 它允许类的使用者来访问它。

(2) `private` (私有成员, 包括类的属性和方法): 只能被该类的成员函数访问, 即只有类的本身能够访问它, 任何类以外的函数对私有成员的访问都是非法的。

(3) `protecte`: 对于派生类来说, 保护成员就像是公有成员, 可以任意访问。但对于程序的其他部分来说, 就像是私有成员不允许访问。

(4) 类中的成员默认是私有的。

## 9.2.2 实例化对象

类是抽象的, 不能赋值, 但是实例化的类即对象就能够赋值了, 而且通过对象可以直接调用函数。实例化对象的形式如图 9-7 所示。

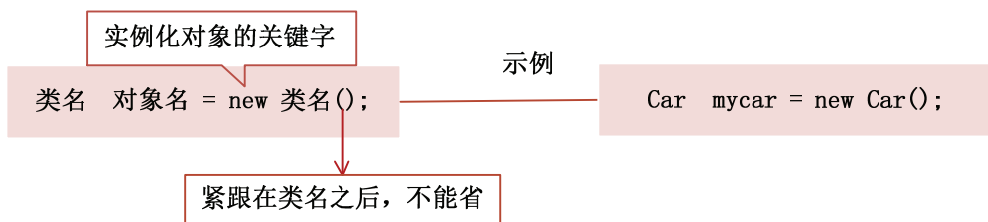


图 9-7 实例化对象



## 9.3 类的属性——数据成员

类的成员中包括类的属性和方法, 本节将简单介绍类的属性。

### 9.3.1 类的属性的定义

类的属性, 又称为数据成员, 用来表示类的信息。类具有的特性均可用属性来表示, 属性的声明方法和变量的声明方法基本相同, 如图 9-8 所示。



图 9-8 类的属性的定义

### 9.3.2 类的数据成员的特例——静态数据成员

静态数据成员是用 `static` 修饰的数据成员。静态数据成员是一种特殊的属性, 在定义类对



象时，不会为每个类对象复制一份静态数据成员，而是让所有的类对象都共享一份静态数据成员备份。其定义的一般形式如图 9-9 所示。

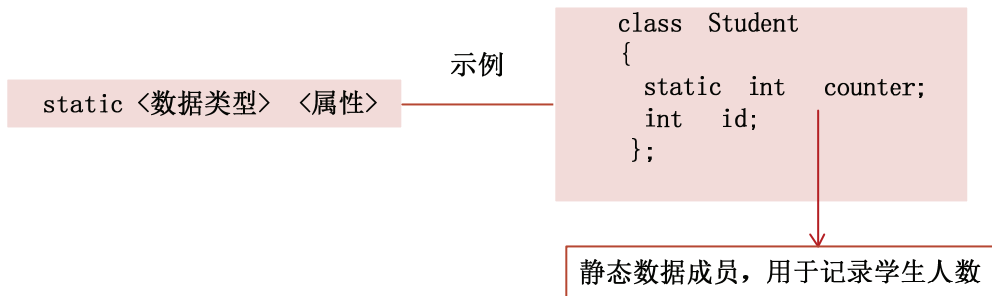


图 9-9 静态数据成员的定义形式

静态变量在定义时要进行初始化，虽然大多数编译器对静态变量会赋默认值，但还是应该自己在对变量定义时就进行初始化。



## 9.4 类的方法——成员函数

类的方法是类的成员的组成部分，本节将详细讲解类的一般定义方法。

### 9.4.1 类的方法的定义

类的方法，又称为类的成员函数。成员函数主要处理类的数据成员，其声明方法和普通函数的声明方法相同，如图 9-10 所示。

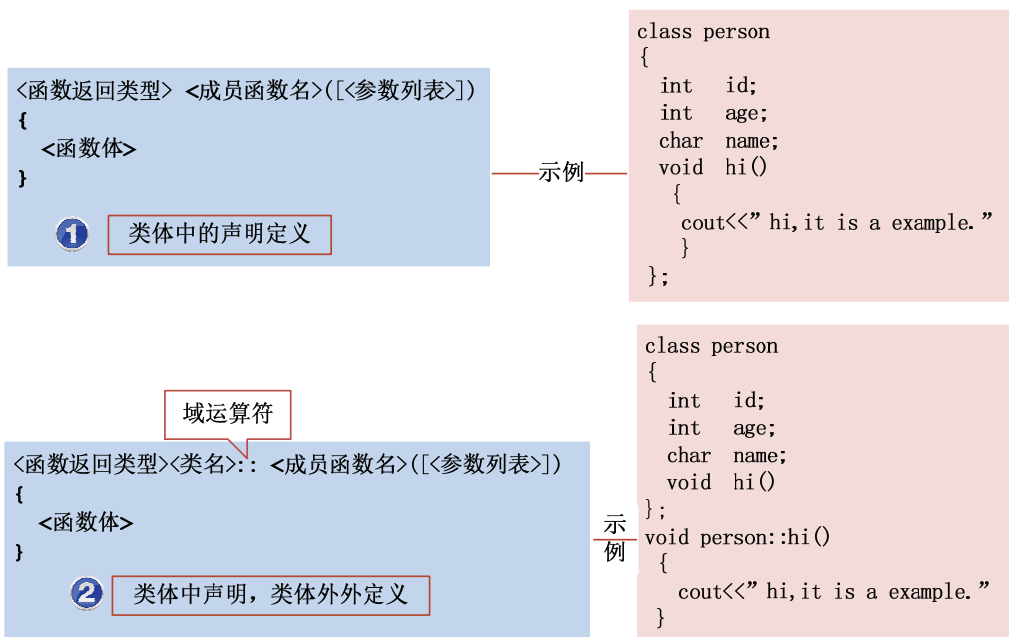


图 9-10 类的方法的定义



## 9.4.2 静态成员函数

静态成员函数是用 `static` 修饰的成员函数。被定义为静态的函数，在类的各个实例化间是共享的，不会为每个类的实例都创建一个静态成员函数的实现。其声明的一般形式如图 9-11 所示。

```
static <函数返回类型> <成员函数名>([<参数列表>])
```

示例

```
class Student
{
    static void setcounter(int);
    void show();
};
```

图 9-11 静态成员函数的声明形式

## 9.4.3 成员函数的类别（const 的另一种用法）

把保留字 `const` 放在函数的参数列表和函数体之间，称为 `const` 成员函数。`const` 成员函数只能读类数据成员，而不能修改类成员数据。如果 `const` 成员函数试图以任何改变类的数据成员或调用另一个非 `const` 成员函数，编译器将给出错误信息。其基本定义格式如下：

(1) 类内定义时：

```
类型 函数名(参数列表) const
{
    函数体
}
```

(2) 类外定义时：

```
类型 类名::函数名(参数列表) const
{
    函数体
}
```



**注意：**类外定义时同样要加关键字 `const`，否则编译器会把其看成一个不同的函数。



## 9.5 特殊的成员函数——构造函数和析构函数

在定义类后，经常需要一一为对象的成员变量指定初始值，而构造函数用于在定义了类的对象数组后进行初始化。

### 9.5.1 构造函数的概念

类包含数据成员和成员函数。一个类可以创建多个对象。创建对象后，需要为每个对象的数据成员赋初值。一一为多对象的多个数据成员赋初值是非常烦琐的，并且在类中声明数据成员，是不给数据成员赋初值的。如图 9-12 中给数据成员赋值是错误的。



```
class A
{
    int a=0;
};
```

错误，此处不能给数据成员a赋值

图 9-12 在声明类时给数据成员赋初值是错误的

因此，在 C++ 中，通常使用构造函数来解决这个问题。构造函数的定义和主要功能如图 9-13 所示。

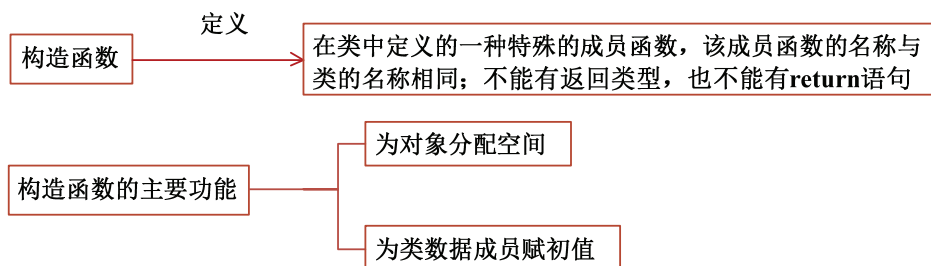


图 9-13 构造函数的定义和主要功能

构造函数相当于要停车就必须找个车位，这个寻找车位的操作就是构造函数所要进行的操作，如图 9-14 所示。

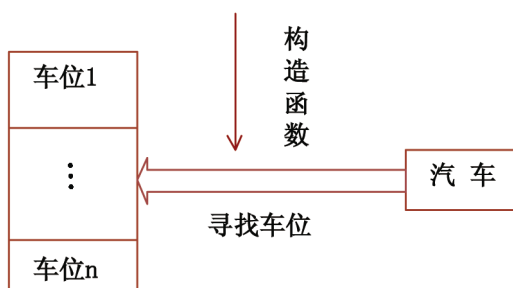


图 9-14 构造函数的形象概念

构造函数的性质如图 9-15 所示。

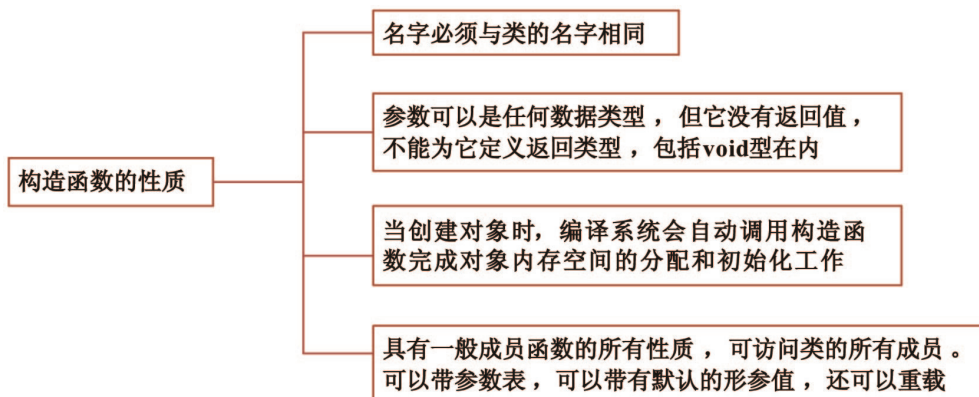


图 9-15 构造函数的性质



## 9.5.2 构造函数的声明和定义

构造函数的声明和定义与普通成员函数的声明和定义类似，如图 9-16 所示。

**1** 类体中声明并定义构造函数

```
class Rectangle
{
    double length, width;
    Rectangle (double a, double b)
    {
        length = a;
        width = b;
    }
    void area ()
    {
        cout<< "Area of Rectangle is : "<<length * width <<endl;
    }
};
```

**2** 类体中声明，类体外定义构造函数

```
class Rectangle
{
    double length, width;
    Rectangle (double a, double b);
    void area ()
    {
        cout<< "Area of Rectangle is : "<<length * width <<endl;
    }
};
Rectangle::Rectangle (double a, double b)
{
    length = a;
    width = b;
}
```

图 9-16 构造函数的声明和定义

实际应用中，一般都要给类声明和定义构造函数。如果没有声明和定义，编译系统就自动生成一个默认的构造函数，这个默认的构造函数不带任何参数，只能给对象开辟一个存储空间，而不能为对象中的数据成员赋初值。此时数据成员的值是随机的，程序运行时可能会出错。系统自动生成构造函数的形式，如图 9-17 所示。

类名::构造函数名()	示例
{ }	Rectangle::Rectangle () { }

图 9-17 系统自动生成的构造函数



**提示：**给对象赋初值是非常重要的工作。



### 9.5.3 构造函数的调用

在定义了构造函数后就可以调用了。一般，在定义对象的同时调用构造函数，其调用的一般形式如图 9-18 所示。

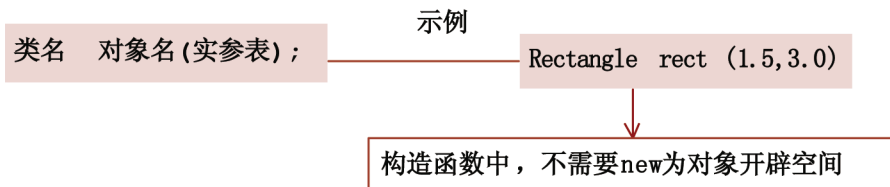


图 9-18 构造函数调用的一般形式



**注意：**构造函数一般不需要显式调用，在声明对象时系统会自动调用构造函数。

【示例 9-1】下面的程序在创建类的同时会自动调用构造函数，实现代码及结果如图 9-19 所示。

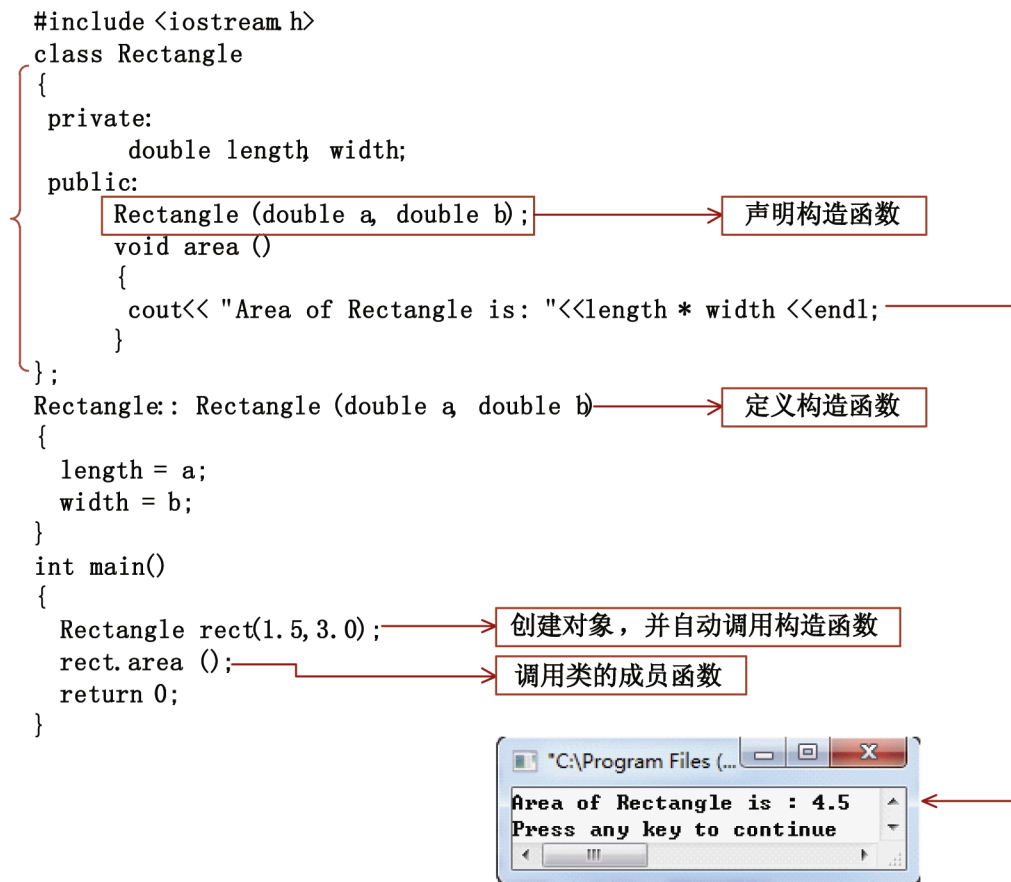


图 9-19 构造函数调用实例



## 9.5.4 不带参数的构造函数

构造函数一般需要完成对类中私有数据成员的初始化，因此它需要包含参数，以便在创建对象时完成对象的初始化。事实上，构造函数可以不带参数。

【示例 9-2】下面程序的构造函数没有带参数，其实现代码及结果如图 9-20 所示。

```
#include <iostream h>
class Myclass
{
private:
    int a;
public:
    Myclass();
    void disp()
    {
        cout<<"a*a = "<<a*a<<endl;
    }
};
Myclass:: Myclass ()
{
    cout<<"initialized\n";
    a=10;
}
int main()
{
    Myclass s;
    s.disp();
    return 0;
}
```

Diagram annotations:

- `Myclass();` → 声明不带参数的构造函数
- `void disp()` → 定义成员函数
- `Myclass:: Myclass ()` → 定义构造函数
- `Myclass s;` → 创建对象，自动执行构造函数
- `s.disp();` → 调用成员函数

图 9-20 不带参数的构造函数实例

不带参数的构造函数对象的初始化是固定的，如希望在建立对象时通过参数初始化数据成员，应使用带参数的构造函数。

## 9.5.5 带有默认参数的构造函数

若构造函数带有参数，在定义对象时必须给构造函数传递参数，否则构造函数将不被执行。在实际应用中，有些构造函数的参数值通常是不变的，只有在特殊情况下才需要改变它的值。

这时，可以将构造函数定义成带默认参数值的构造函数，这样，在定义对象时可以不指定实参，用默认参数值来初始化数据成员。

【示例 9-3】下面的程序调用的构造函数带有默认参数，其实现代码及结果如图 9-21 所示。



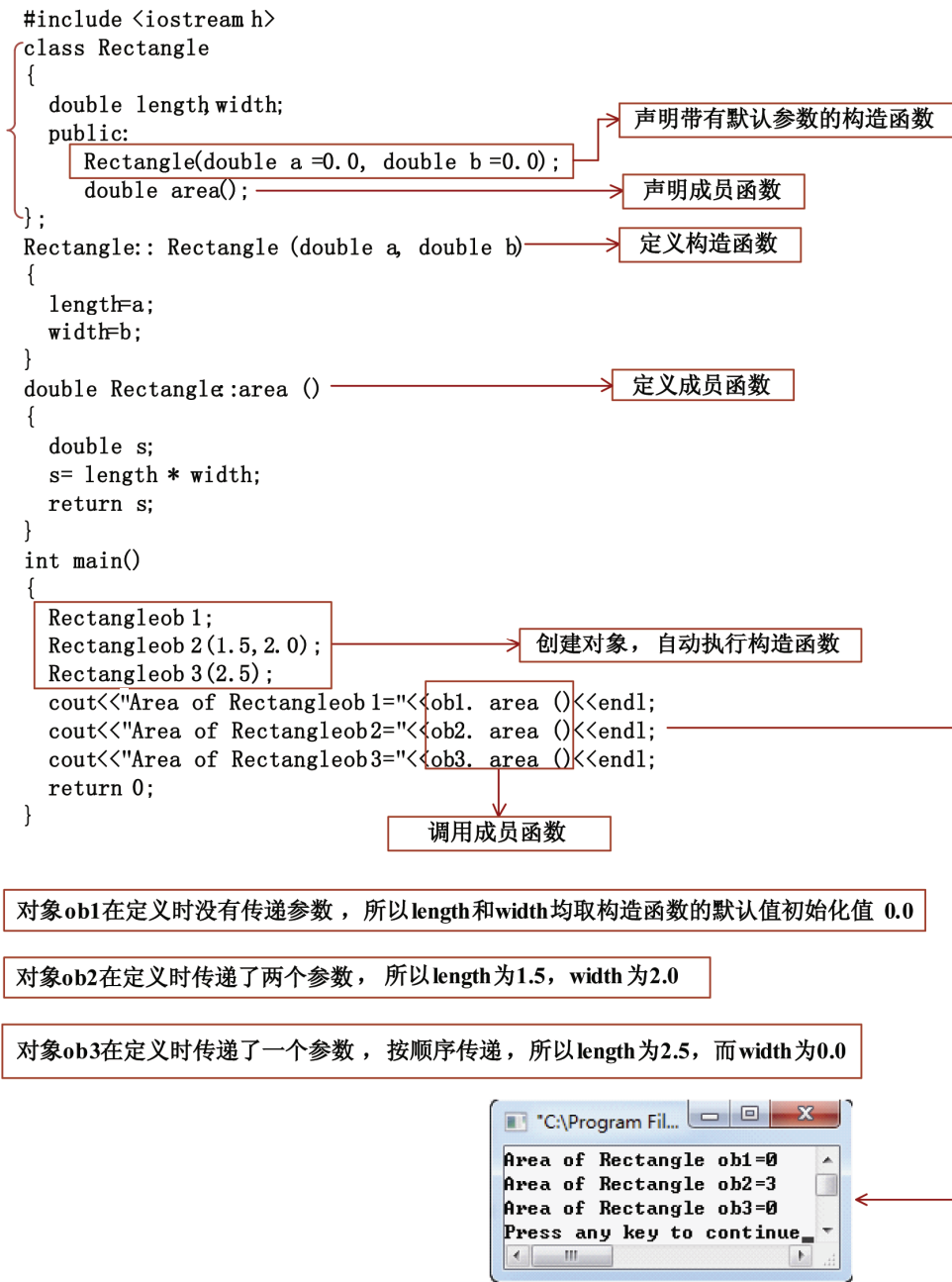


图 9-21 带有默认参数的构造函数实例

### 9.5.6 构造函数的重载

C++支持函数的重载，一个类中也可以有多个不同参数形式的构造函数。用类去创建一个对象（后面可以附带参数），也就是在内存中产生一个类的实例时，程序将根据参数自动调用该类中对应的构造函数。

对构造函数进行重载可以适应不同的情况，增加了程序设计的灵活性，这些构造函数的区分依据如图 9-22 所示。



构造函数的区分依据

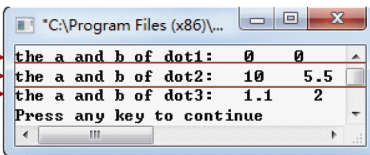
参数的个数

参数的类型

图 9-22 构造函数的区分依据

【示例 9-4】下面的程序通过对构造函数进行重载来实现程序的灵活性，其实现代码及结果如图 9-23 所示。

```
#include <iostream.h>
class Dot
{
private:
    double a, b;
public:
    Dot();
    Dot(double x, double y);
    void showDot();
};
Dot::Dot ()
{
    a=0.0;
    b=0.0;
}
Dot::Dot(double x, double y=5.5)
{
    a=x;
    b=y;
}
void Dot::showDot()
{
    cout<<a<<"    "<<b<<endl;
}
int main()
{
    Dot dot1;
    cout<<"the a and b of dot1: ";
    dot1.showDot();
    Dot dot2(10);
    cout<<"the a and b of dot2: ";
    dot2.showDot();
    Dot dot3(1.1, 2.0);
    cout<<"the a and b of dot3: ";
    dot3.showDot();
    return 0;
}
```



Dot dot1;语句创建dot1对象时，调用的是Dot()构造函数

Dot dot2(10);语句创建dot2对象时，调用的是Dot (double x,double y)构造函数，这时只有一个实参，则另一个y为默认值5.5

在语句Dot dot3 (1.1,2.0);中调用的是Dot (double x,double y)构造函数，其y的默认值被重新赋值为2.0。

图 9-23 构造函数的重载实例



**注意：**当定义带参数的构造函数时采用了默认的参数，在调用时若指定了参数值，则使用该指定值，否则采用默认值。

### 9.5.7 特殊的构造函数——复制构造函数

在 C++ 中，除了普通的构造函数外，还有一类特殊的构造函数——复制构造函数。复制构造函数的作用和分类如图 9-24 所示。

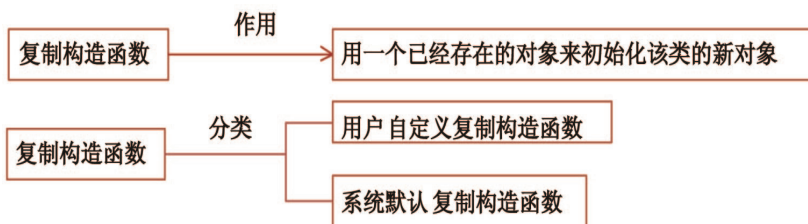


图 9-24 复制构造函数的作用和分类

自定义复制构造函数的形式如图 9-25 所示。

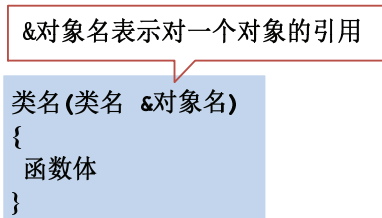
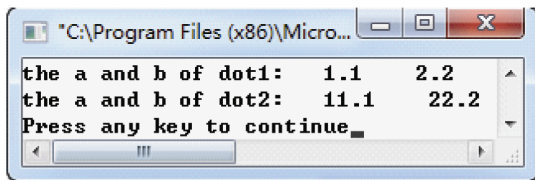


图 9-25 自定义拷贝构造函数的形式

**【示例 9-5】**下面的程序实现一个用户自定义的复制构造函数，其实现代码及结果如图 9-26 所示。



```
#include <iostream h>
class Dot
{
private:
    double a, b;
public:
    Dot (Dot &d); // 声明自定义复制构造函数
    Dot (double x, double y); // 声明构造函数
    void showDot(); // 声明成员函数
};
Dot::Dot(Dot &d) // 定义自定义复制构造函数
{
    a=d.a+10;
    b=d.b+20;
}
Dot:: Dot(double x, double y) // 定义构造函数
{
    a=x;
    b=y;
}
void Dot::showDot() // 定义成员函数
{
    cout<<a<<"    "<<b<<endl;
}
int main()
{
    Dot dot1(1.1, 2.2); // 用构造函数创建对象
    cout<<"the a and b of dot1:  ";
    dot1.showDot();
    Dot dot2(dot1); // 用复制构造函数创建对象
    cout<<"the a and b of dot2:  ";
    dot2.showDot();
    return 0;
}
```



Dot dot1(1.1, 2.2); 创建了对象 dot1, 其调用的是构造函数 Dot(double x, double y)

Dot dot2(dot1); 语句创建了对象 dot2, 其调用的是复制构造函数 Dot(Dot &d)

图 9-26 用户自定义的复制构造函数实例

当用一个已经存在的对象初始化本类的新对象时, 如果没有自定义复制构造函数, 则系统会自动生成一个默认的复制构造函数来完成初始化的工作。

【示例 9-6】下面的程序将使用默认的复制构造函数, 其实现代码及结果如图 9-27 所示。



```

#include <iostream h>
class Dot
{
private:
    double a,b;
public:
    Dot (double x,double y);
    void showDot();
};
Dot::Dot(double x,double y)
{
    a=x;
    b=y;
}
void Dot::showDot()
{
    cout<<a<<"    "<<b<<endl;
}
int main()
{
    Dot dot1(1.1, 2.2);
    cout<<"the a and b of dot1: "<<endl;
    dot1.showDot();
    Dot dot2(dot1);
    cout<<"the a and b of dot2: "<<endl;
    dot2.showDot();
    return 0;
}

```

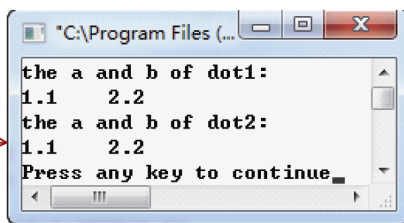


图 9-27 使用默认的复制构造函数实例

### 9.5.8 析构函数

与构造函数类似，析构函数也是一种特殊的成员函数。析构函数作用如图 9-28 所示。



图 9-28 析构函数的作用

在声明、定义和使用析构函数时需要注意的事项如图 9-29 所示。

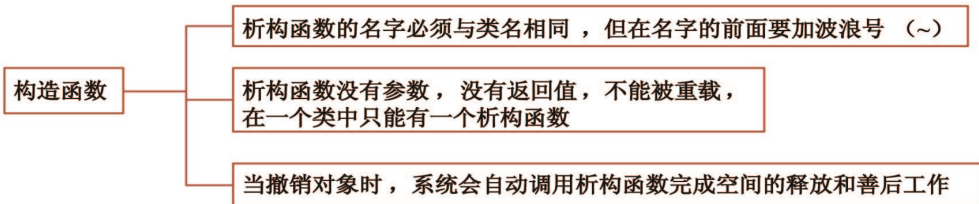


图 9-29 声明、定义和使用析构函数时需要注意的事项



【示例 9-7】下面的程序创建类 Rectangle，在主程序中创建该类的对象，当程序结束时程序调用析构函数。其实现代码及结果如图 9-30 所示。

```
#include <iostream h>
class Rectangle
{
    double length,width;
public:
    Rectangle(double length=0.0, double width=0.0);
    ~ Rectangle();
    double area();
};
Rectangle::Rectangle(double x,double y)
{
    cout<<"constructing....."<<endl;
    length = x;
    width = y;
}
Rectangle::~~ Rectangle()
{
    cout<<"destructing....."<<endl;
}
double Rectangle::area()
{
    return length*width;
}

int main()
{
    Rectangle ob(1.1, 2.2);
    cout<<"area of Rectangle ob="<<ob.area()<<endl;
    return 0;
}
```

Diagram annotations for the code:

- `Rectangle(double length=0.0, double width=0.0);` → 声明构造函数
- `~ Rectangle();` → 声明析造函数
- `double area();` → 声明成员函数
- `Rectangle::Rectangle(double x,double y)` → 定义构造函数
- `Rectangle::~~ Rectangle()` → 定义析造函数
- `double Rectangle::area()` → 定义成员函数
- `Rectangle ob(1.1, 2.2);` → 调用构造函数创建对象
- `return 0;` → 释放对象调用析构函数

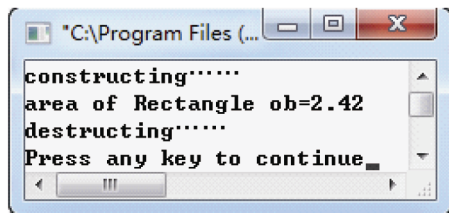


图 9-30 析构函数实例

### 9.5.9 类和函数的联系

类和函数有如下联系：

- (1) 类对象可以作为函数的参数，也可以作为函数的返回值。
- (2) 类对象作为函数参数传递时，可以值传递和引用传递。
- (3) 类对象值传递时，实参中的成员变量目录中的变量被复制到形式参数中相应的成员变量。



### 9.5.10 this 指针

this 指针是隐含在成员函数内的一种指针，称为指向本对象的指针，可以采用如 this->数据成员的方式来存取类的数据成员。

this 指针主要有如下作用：

- (1) 显示指明类中的数据成员，尤其是和形参及全局变量相区分。
- (2) 返回本对象的指针和引用。

在函数成员返回时，如果需要返回该对象的指针，只需使用“return this”；在返回本对象的引用时只需使用“return \*this”。

- (3) 类对象值传递时，实参中的成员变量被复制到形式参数中相应的成员变量。



## 9.6 小结

类是 C++ 中最基本的特征，允许定义新的类型以适应程序的需要。本章介绍了如何定义类，以及如何创建对象来使用类。重点讲解了类的属性和方法及特殊的成员函数——构造函数与析构函数。



## 9.7 习题

【题目 9-1】定义一个电脑类，它有品牌和价格两个数据成员。它的方法有输出、设置价格、设置品牌 3 个方法。要求：编写类的方法时，分别在类内和类外两种形式编写。程序的运行结果如图 9-31 所示。

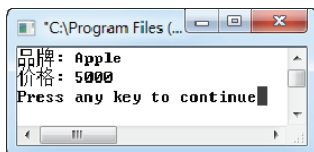


图 9-31 运行结果

【题目分析】本题主要考查类的声明、实例化对象，以及如何用对象调用成员函数。

【关键代码】

类内定义 computer 类的成员函数：

```
class computer
{
private:
    char brand[20];
    double price;
public:
    void print()
    {
        cout<<"品牌: "<<brand<<endl;
        cout<<"价格: "<<price<<endl;
    }
    void SetBrand(char *sz)
    {
        strcpy(brand,sz);
    }
    void SetPrice(double pr)
```



```
        {
            price=pr;
        }
    };

int main()
{
    computer com1;
    com1.SetPrice(5000);
    com1.SetBrand("Apple");
    com1.print();
    return 0;
}
```

类外定义成员函数:

```
class computer
{
private:
    char brand[20];
    double price;
public:
    void print();
    void SetBrand(char *sz);
    void SetPrice(double pr);
};

void computer::print()
{
    cout<<"品牌: "<<brand<<endl;
    cout<<"价格: "<<price<<endl;
}

void computer::SetBrand(char *sz)
{
    strcpy(brand,sz);
}

void computer::SetPrice(double pr)
{
    price=pr;
}
```

【题目 9-2】定义一个三维坐标点的类，分别定义有参和无参的构造函数。在主函数中实例化对象时分别使用它们。程序的运行结果如图 9-32 所示。

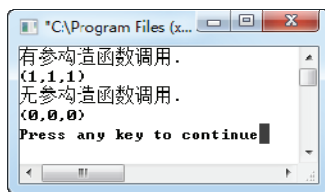


图 9-32 运行结果

【题目分析】本题主要考查构造函数的创建、调用、重载等知识。

【关键代码】

```
class point
{
private:
    int xPos;
    int yPos;
    int zPos;
public:
```





```

point()
{
    cout<<"无参构造函数调用."<<endl;
    xPos=0;
    yPos=0;
    zPos=0;
}
point(int x,int y,int z)
{
    cout<<"有参构造函数调用."<<endl;
    xPos=x;
    yPos=y;
    zPos=z;
}
print()
{
    cout<<"("<<xPos<<","<<yPos<<","<<zPos<<") "<<endl;
}

};

int main()
{
    point p1(1,1,1);
    p1.print();
    point p2;
    p2.print();
    return 0;
}

```

**【题目 9-3】** 再在【题目 9-2】的基础上写一个带有默认参数的构造函数，构造函数按参数默认方式调用。

**【题目分析】** 本题主要考查带参数的构造函数的定义方法和调用方法。

**【关键代码】**

```

point(int x,int y,int z)
{
    cout<<"带默认参数的构造函数调用."<<endl;
    xPos=x;
    yPos=y;
    zPos=z;
}

```

**【题目 9-4】** 在【题目 9-2】的基础上，再自定义一个复制构造函数，不允许使用编译器默认的复制构造函数。程序运行结果如图 9-33 所示。

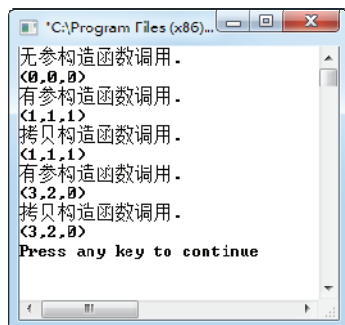


图 9-33 运行结果



【题目分析】 本题主要考查复制构造函数的使用。

【关键代码】

```
point(const point &pt)
{
    cout<<"复制构造函数调用."<<endl;
    xPos=pt.xPos;
    yPos=pt.yPos;
    zPos=pt.zPos;
}
```

【题目 9-5】 定义一个计算机类，它包含品牌、价格两个属性。自定义一个构造方法和一个析构方法，在定义构造方法时，品牌数据采用动态分配存储空间的方式申请空间。在定义的析构函数中，输出“现场清理完毕”的提示信息。另外，再定义一个输出函数输出计算机的信息。在主函数中，实例化两个对象，并用对象调用输出函数输出计算机的品牌和价格。然后分别用隐式和显式方式调用析构函数，清理现场。程序的运行结果如图 9-34 所示。

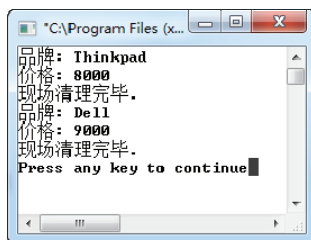


图 9-34 运行结果

【题目分析】 本题主要考查析构函数的定义方法及怎样调用析构函数。

【关键代码】

```
class computer
{
private:
    char *brand;
    double price;
public:
    computer(const char *sz,double p)
    {
        brand=new char[strlen(sz)+1];
        strcpy(brand,sz);
        price=p;
    }
    ~computer()
    {
        delete [] brand;
        cout<<"现场清理完毕."<<endl;
    }
    void print()
    {
        cout<<"品牌: "<<brand<<endl;
        cout<<"价格: "<<price<<endl;
    }
};

int main()
{
    computer com1("Thinkpad",8000);
    com1.print();
```



```
com1.~computer();
computer com2("Dell",9000);
com2.print();
```

【题目 9-6】定义一个计算机类，它的属性有品牌、售价和总价。编写程序，统计每天卖出计算机的总价钱。要求：将总价定义为 `static` 类型，程序可以根据每天销售出去的计算机台数和退货的情况计算出销售总价。程序运行结果如图 9-35 所示。

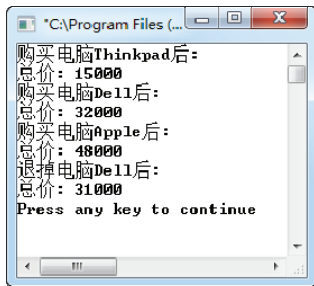


图 9-35 运行结果

【题目分析】本题主要考查类的特殊数据成员——静态数据成员的用法。静态数据类型是所有对象共享的数据成员。

#### 【关键代码】

```
class computer
{
private:
    char *brand;
    double price;
    static double total;
public:
    computer(char *sz,double p)
    {
        brand=sz;
        price=p;
        total=total+price;
    }
    ~computer()
    {
        total=total-price;
    }
    void print1()
    {
        cout<<"总价: "<<total<<endl;
    }
};

double computer::total=0;
int main()
{
    computer com1("Thinkpad",15000);
    cout<<"购买电脑 Thinkpad 后: "<<endl;
    com1.print1();
    computer com2("Dell",17000);
    cout<<"购买电脑 Dell 后: "<<endl;
    com2.print1();
    computer com3("Apple",16000);
    cout<<"购买电脑 Apple 后: "<<endl;
    com3.print1();
    com2.~computer();
```



```
        cout<<"退掉电脑 Dell 后: "<<endl;
        com3.print1();

        return 0;
    }
```

【题目 9-7】在【题目 9-6】的基础上，定义静态函数，输出总价。程序运行结果如图 9-36 所示。

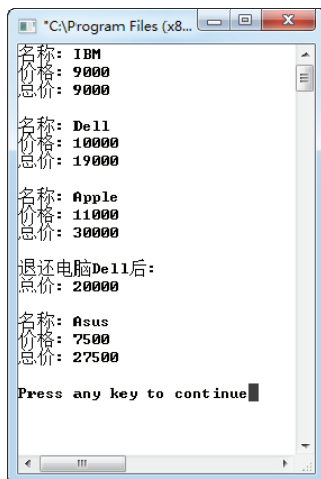


图 9-36 运行结果

【题目分析】本题主要考查静态函数的定义和使用方法。静态函数也是所有对象共享的，调用时用类名直接调用。

#### 【关键代码】

```
class computer
{
private:
    char *brand;
    double price;
    static double total;
public:
    computer(char *sz,double p)
    {
        brand=sz;
        price=p;
        total=total+price;
    }
    ~computer()
    {
        total=total-price;
    }
    static void print_total()
    {
        cout<<"总价: "<<total<<endl<<endl;
    }
    void print()
    {
        cout<<"名称: "<<brand<<endl;
        cout<<"价格: "<<price<<endl;
    }
};
double computer::total=0;
```



```
int main()
{
    computer com1("IBM",9000);
    com1.print();
    computer::print_total();
    computer com2("Dell",10000);
    com2.print();
    computer::print_total();
    computer com3("Apple",11000);
    com3.print();
    computer::print_total();
    com2.~computer();
    cout<<"退还电脑 Dell 后:"<<endl;
    computer::print_total();
    computer com4("Asus",7500);
    com4.print();
    computer::print_total();

    return 0;
}
```

【题目 9-8】定义一个学生类，包括数据成员为年龄、性别、身高、成绩及显示成员函数。最后声明一个该类的对象，调用相应的显示函数输出结果。程序运行结果如图 9-37 所示。

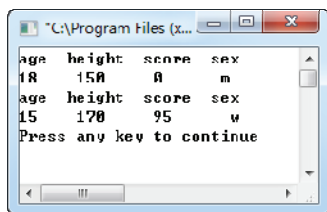


图 9-37 运行结果

【题目分析】本题考查类和对象的基本知识，重点是掌握类中各种成员的声明、定义和使用。

#### 【关键代码】

```
class student
{
private:
    int age;
    int height;
    int score;
    char sex;
public:
    student()
    {
        sex='m';
        age=18;
        score=0;
        height=150;
    }
    student(int a,int h,int s,char x)
    {
        age=a;
        height=h;
        score=s;
        sex=x;
    }
    void display()
    {
```



```
        cout<<"age height score sex"<<endl;
        cout<<age<<"    "<<height<<"    "<<score<<"    "<<sex<<endl;
    }
};
```

【题目 9-9】定义一个包含复制构造函数的学生类，先定义一个该类的对象，然后用该对象初始化另外一个对象，最后输出结果。运行结果如图 9-38 所示。

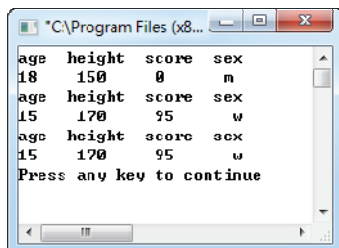


图 9-38 运行结果

【题目分析】本题主要考查复制构造函数的掌握情况，重点是掌握显式定义复制构造函数和调用方法。

【关键代码】

```
class student
{
private:
    char *name;
    int age;
    int height;
    int score;
    char sex;
public:
    student()
    {
        sex='m';
        age=18;
        score=0;
        height=150;
    }
    student(int a,int h,int s,char x)
    {
        age=a;
        height=h;
        score=s;
        sex=x;
    }
    student(const student &stu)
    {
        age=stu.age;
        height=stu.height;
        score=stu.score;
        sex=stu.sex;
    }
    void display()
    {
        cout<<"age height score sex"<<endl;
        cout<<age<<"    "<<height<<"    "<<score<<"    "<<sex<<endl;
    }
};

student stu3(stu2);
```

# 第 10 章 继承与派生

类的继承是面向对象的重要特征，C++提供了丰富的继承功能。通过继承和派生，可以简化代码的编写，提高开发效率。本章将详细介绍有关继承与派生的知识。

## 10.1 继承与派生的基础语法

继承是面向对象的一块基石，是允许创建分等级层次类的关键。因为有继承，可以创建一个通用的类，然后派生出多个类，在这些类里可以增加新的成员，以实现具体、多样的功能。

### 10.1.1 继承与派生概述

类的继承是新类从已有类那里获得特性，从已有的类产生新类的过程称为类的派生。已有类称为基类或父类，派生出的新类则称为派生类或子类。

继承的功能主要体现在两个方面，如图 10-1 所示。

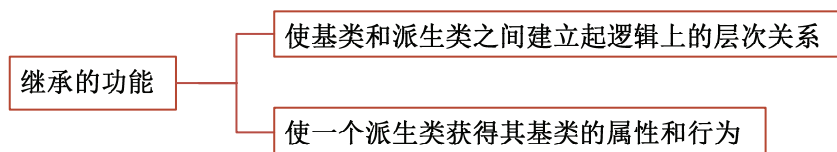


图 10-1 继承的功能

在生活中，有许多继承的例子。例如，人可以看做一个类，动物也是一个类，而人这个类具有动物这个类的所有特征，同时还具有其他动物所没有的特征，如说话、直立行走等。因此，在该关系中，人是派生类，动物是基类，人这个类继承自动物类。

根据派生类所拥有的基类数目不同，可以分为单继承和多继承，如图 10-2 所示。

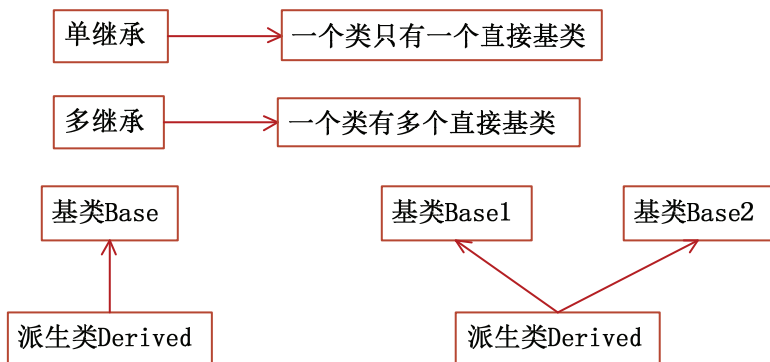


图 10-2 单继承和多继承

基类与派生类之间的关系如图 10-3 所示。

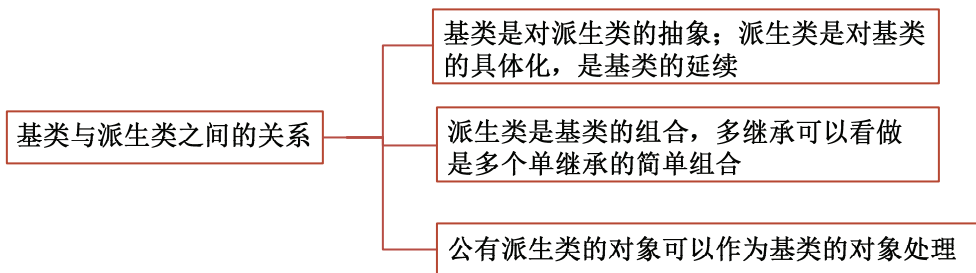


图 10-3 基类与派生类之间的关系

### 10.1.2 声明派生类

派生类声明的一般形式如图 10-4 所示。

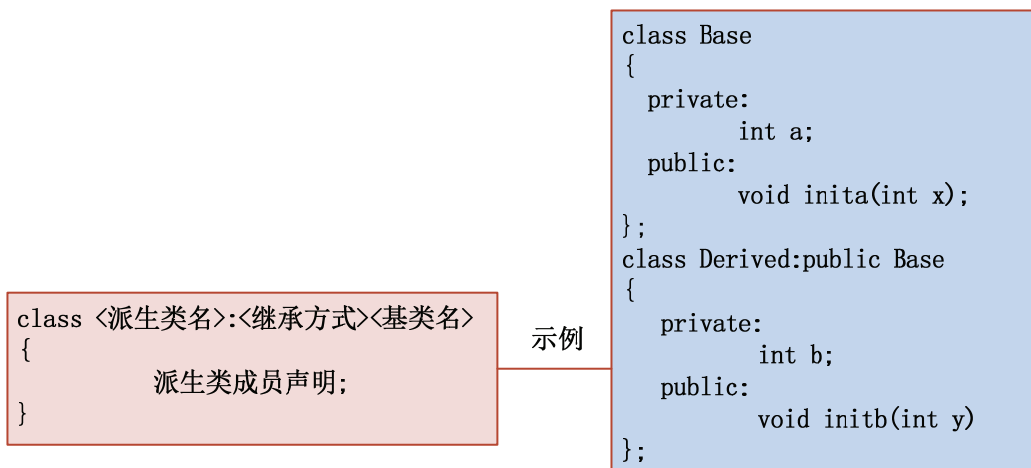


图 10-4 派生类声明的一般形式





## 10.2 成员的访问

访问说明符用来控制相应成员在程序中的可访问性，使得信息封装和模块化的风格更好。本节将详细介绍 C++ 常用的访问说明符及成员的访问权限。

### 10.2.1 类的成员的访问说明符

在 C++ 中，类的成员被分成 3 类，如图 10-5 所示。

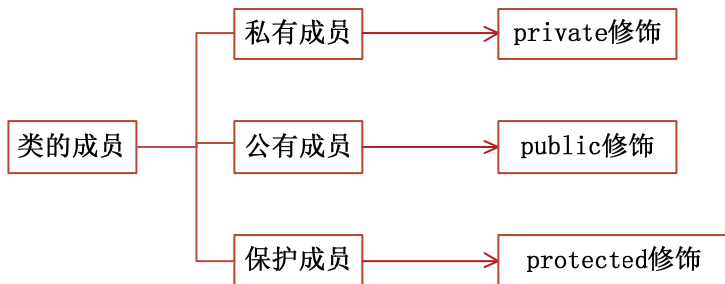


图 10-5 类的成员分类



**提示：**如果没有明确声明，则默认为私有成员。

### 10.2.2 类的成员的访问权限

成员被不同的说明符修饰，其访问权限是不同的，如图 10-6 所示。

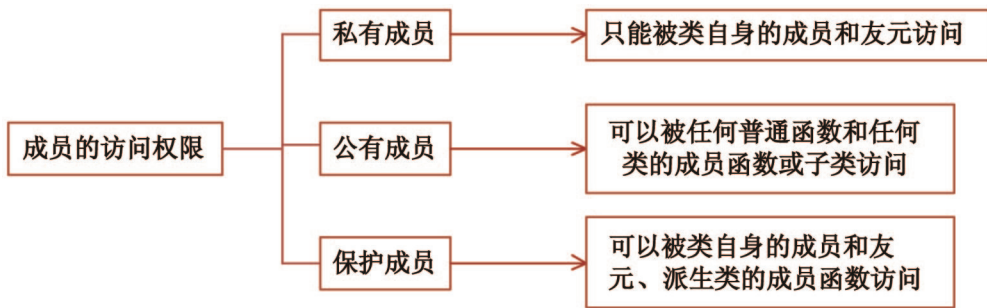
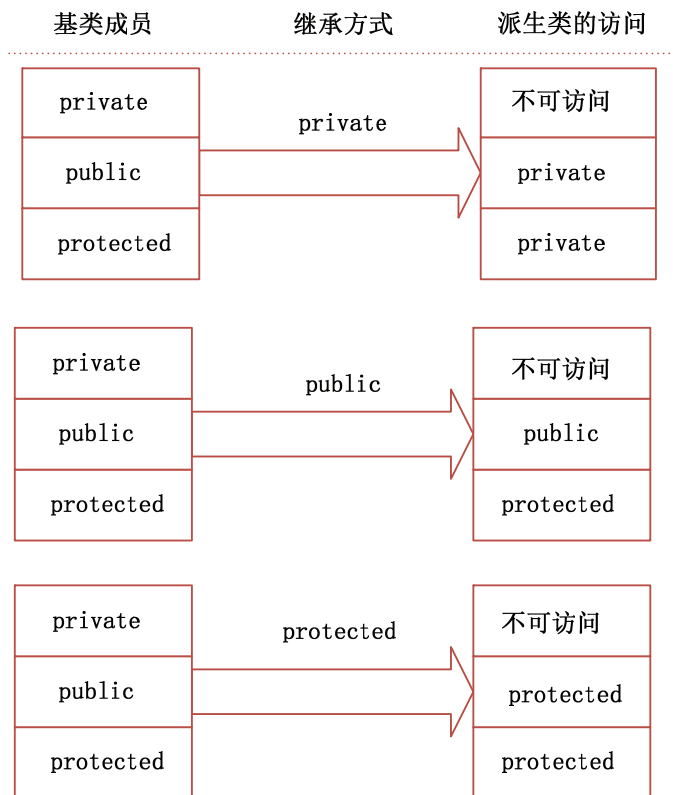


图 10-6 成员的访问权限

## 10.3 继承的访问控制

类的继承方式有公有继承（public）、保护继承（protected）和私有继承（private）3 种，不同的继承方式，导致派生类对基类成员的访问能力有所不同。总的来说，派生类对基类成员的访问能力如图 10-7 所示。



- 1 基类中的私有成员在派生类中是隐藏的，只能在基类内部访问
- 2 派生类中的成员不能访问基类中的私有成员，但可访问基类中的公有成员和保护成员
- 3 派生类从基类私有继承时，基类的公有和保护成员在派生类中都改为私有成员
- 4 派生类从基类公有继承时，基类的公有和保护成员在派生类中仍为公有和保护成员
- 5 派生类从基类保护继承时，基类的公有成员在派生类中改变为保护成员，保护成员不变

图 10-7 派生类对基类成员的访问能力

### 10.3.1 私有继承

在私有继承中，派生类以私有方式继承基类。派生类不能直接访问基类的私有成员，而只能在派生类的成员函数中通过基类的公有或保护成员函数间接访问。在设计基类时，通常都要为其私有成员提供能够访问它们的公有成员函数，以便派生类和外部函数能间接访问它们。

【示例 10-1】下面的程序以私有继承的方式继承基类，此时基类的公有成员和保护成员在派生类中的属性有所变化，需仔细理解。其实现代码及结果如图 10-8 所示。



```

#include <iostream h >
class Base
{
private:
    int a;
public:
    void inita(int x)
    {
        a=x;
    }
    int geta()
    {
        return a;
    }
};

class Derived:private Base
{
private:
    int b;
public:
    void initb(int y,int x)
    {
        b=y;
        inita(x);
    }
    int getb()
    {
        return b+geta();
    }
};

int main()
{
    Derived ob;
    ob.initb(5,7);
    cout<<ob.getb()<<endl;
    return 0;
}

```

定义基类

定义基类私有成员

定义基类公有成员

类Derived以私有方式继承类Base

调用基类成员函数

创建派生类对象

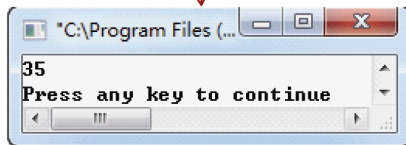


图 10-8 私有继承实例



**注意：**在实际应用中，由于基类经过多次派生后，其私有成员可能会成为不可访问的，所以用得比较少。

### 10.3.2 公有继承

在公有继承中，基类的公有成员和保护成员在派生类中仍是公有成员和保护成员，派生类的成员函数可直接访问它们，而外部函数只能通过派生类的对象间接访问它们。



【示例 10-2】以下程序以公有继承的方式继承基类，通过类外的对象访问基类的公有成员。其实现代码及结果如图 10-9 所示。

```
#include <iostream h>
class Base
{
private:
    int a;
public:
    void inita(int x)
    {
        a=x;
    }
    int geta()
    {
        return a;
    }
};
class Derived:public Base
{
private:
    int b;
public:
    void initb(int y)
    {
        b=y;
    }
    int getb()
    {
        return b*geta();
    }
};
int main()
{
    Derived ob;
    ob.inita(12);
    ob.initb(5);
    cout<<"the result of ob.getb() is: "<<ob.getb()<<endl;
    return 0;
}
```

Annotations for the code:

- 定义基类 (Define Base Class) points to `class Base`
- 定义基类私有成员 (Define Base Class Private Members) points to `private: int a;`
- 定义基类公有成员 (Define Base Class Public Members) points to `public: void inita(int x) ... int geta() ...`
- 类Derived以公有方式继承类Base (Class Derived inherits Class Base publicly) points to `class Derived:public Base`
- 调用基类成员函数 (Call Base Class Member Function) points to `return b*geta();`
- 创建派生类对象 (Create Derived Class Object) points to `Derived ob;`
- 通过类外的对象访问基类的公有成员 (Access Base Class Public Members via Object) points to `ob.inita(12);`
- 调用派生类的公有成员 (Call Derived Class Public Members) points to `ob.initb(5);`

Output of the program:

```
the result of ob.getb() is: 60
Press any key to continue
```

图 10-9 公有继承实例



在使用公有继承时要注意如图 10-10 所示的两个问题。

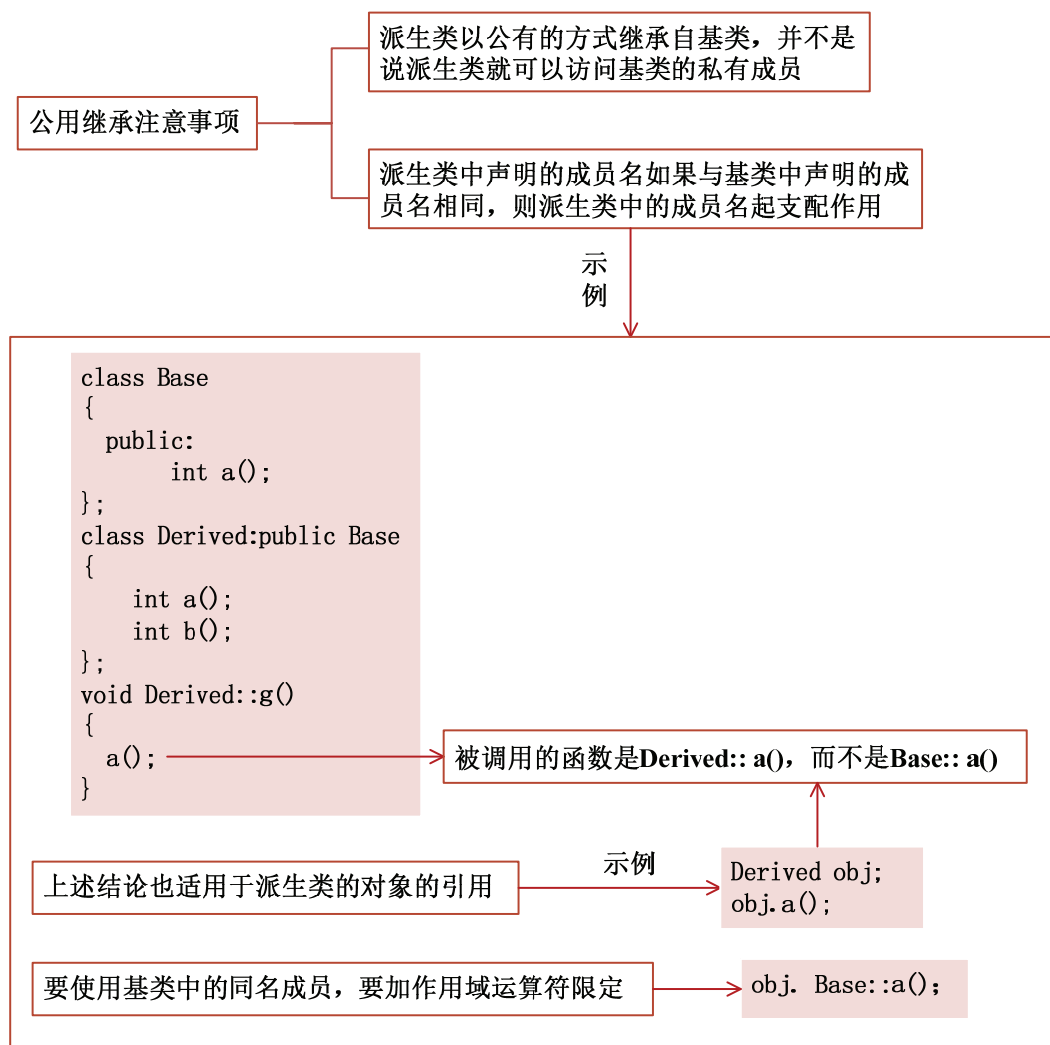


图 10-10 使用公有继承时注意的两个问题

### 10.3.3 保护继承

在保护继承中，基类的公有成员在派生类中称为保护成员，基类的保护成员在派生类中仍为保护成员，所以，派生类的所有成员在类的外部都无法访问它们。

【示例 10-3】下面的程序实现类 `Derived` 对基类 `Base` 的保护继承，仔细理解保护继承后派生类对象如何访问基类的成员。其实现代码及结果如图 10-11 所示。



```
#include<iostream h>
class Base —————> 定义基类
{
private: —————> 定义基类私有成员
    int a;
protected: —————> 定义基类保护成员
    int b;
public: —————> 定义基类公有成员
    int c;
    void setab(int x, int y)
    {
        a=x;
        b=y;
    }
    int geta()
    {
        return a;
    }
};

class Derivedprotected Base —————> 类Derived以保护方式继承类Base
{
private:
    int c;
public:
    void setabc(int m, int n, int l)
    {
        setab(m, n);
        b=m;
        c=l;
    }
    int getc()
    {
        c=c+b*geta();
        return c;
    }
};

int main()
{
    Derived ob; —————> 创建派生类对象

    // ob.setab(12, 12); —————> 非法，不能通过类外对象访问从基类保护继承来的成员

    ob.setabc(12, 12, 5); —————> 合法，访问派生类本身的成员函数

    cout<<"the result of ob.getc() is: "<<ob.getc()<<endl;
    return 0;
}
```

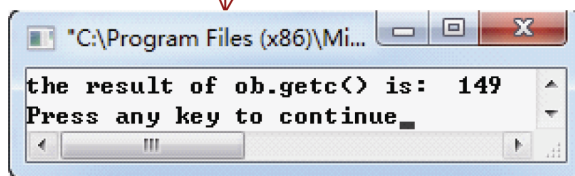


图 10-11 保护继承实例



### 10.3.4 特殊方法的继承——派生类的构造函数和析构函数

在 C++ 中，基类成员的初始化工作由基类的构造函数完成，而派生类的初始化工作由派生类的构造函数完成。

#### 1. 构建原则

由于基类和派生类都需要调用构造函数来实现初始化成员，这就产生了派生类构造函数和析构函数。在构建派生类的构造函数和析构函数时，要遵循的原则如图 10-12 所示。

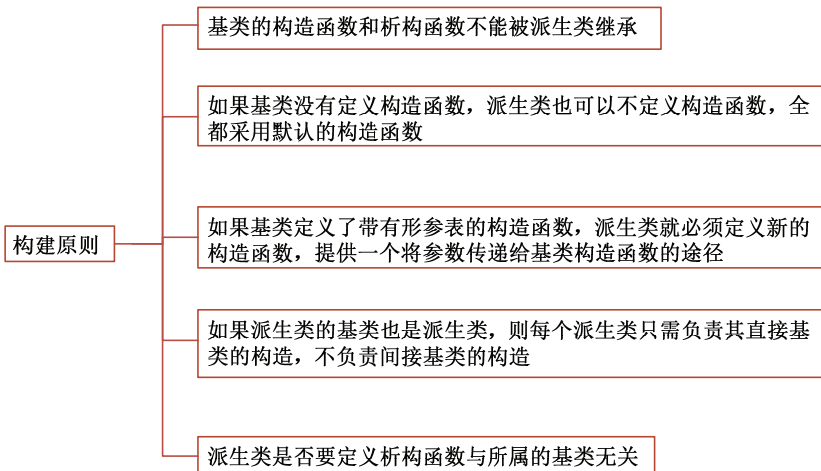


图 10-12 构建派生类的构造函数和析构函数时要遵循的原则

#### 2. 派生类构造函数的创建

派生类的构造函数需要用合适的初值作为参数，隐含调用基类的构造函数和新增对象成员的构造函数来初始化各自的数据成员，再用新加的语句对新增数据成员进行初始化。派生类构造函数声明的一般形式如图 10-13 所示。

```
<构造函数名>(参数总表):基类名(参数表),对象成员名1(参数表1),...,对象成员名n(参数表n)
{
    派生类新增成员的初始化语句
}
```

示例

```
class Base
{
    int i;
    public:
        Base(int n);
}
class Derived:public Base
{
    int j;
    Base ob;
    public:
        Derived(int m): Base(m), ob(m);
}
```

声明基类的构造函数

派生类新增数据成员

基类对象作为派生类对象成员

声明派生类的构造函数

图 10-13 派生类构造函数声明的一般形式



### 3. 派生类析构函数的构建

派生类析构函数的功能与基类析构函数的功能一样，也是在对象撤销时进行必要的内存释放和善后处理工作。析构函数不能被继承，如果需要析构函数，则要在派生类中重新定义。与基类的析构函数一样，派生类的析构函数也没有数据类型和参数。

【示例 10-4】下面的程序定义派生类 Derived，其公有继承于基类 Base，在派生类中构建析构函数。其实现代码及结果如图 10-14 所示。

```
#include<iostream h>
class Base
{
    int i;
    public:
        Base(int n)
        {
            cout<<"constructing Base class\n";
            i=n;
        }
        ~Base()
        {
            cout<<"destructing Base class\n";
        }
        void showi()
        {
            cout<<i<<endl;
        }
};

class Derivedpublic Base
{
    int j;
    Base ob;
    public:
        Derived(int n):Base(n),ob(n)
        {
            cout<<"constrycting Derived class"<<endl;
            j=2*n;
        }
        ~Derived()
        {
            cout<<"destructing Derived class\n";
        }
        void showj()
        {
            cout<<j<<endl;
        }
};

int main()
{
    Derived ob(10);
    ob.showi();
    ob.showj();
    return 0;
}
```

定义基类

基类构造函数

基类析构函数

定义基类成员函数

类Derived以公有方式继承类Base

基类对象作为派生类对象成员

派生类构造函数

派生类析构函数

定义派生类成员函数

创建派生类对象，系统自动调用derived类的构造函数

调用基类成员函数

调用派生类成员函数

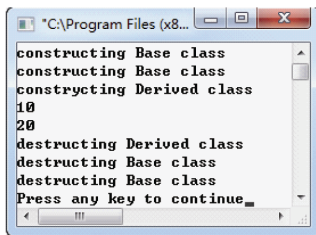


图 10-14 派生类析构函数实例





在【示例 10-4】中两个类都定义了构造函数和析构函数，从如图 10-14 所示的结果中可以看出其执行顺序，如图 10-15 所示。

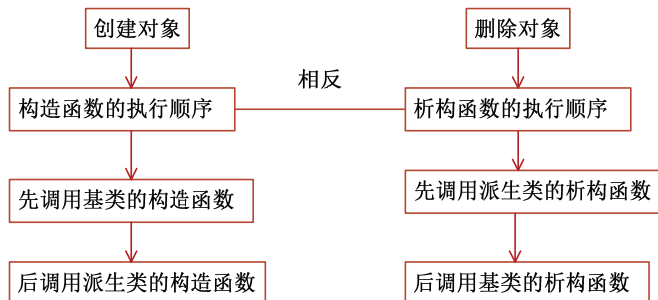


图 10-15 执行顺序



## 10.4 多重继承

以上介绍的继承都是针对派生类只有一个基类的情况，这种情况称为单一继承。而当一个派生类具有多个基类时，称这种派生为多重继承。本节将详细介绍多重继承及其二义性问题。

### 10.4.1 声明多重继承

多重继承就是一个类继承多个基类，那么这个类具有多个基类的属性和行为，同时还可以定义新的属性和行为。

在 C++ 中，多重继承声明的一般形式如图 10-16 所示。

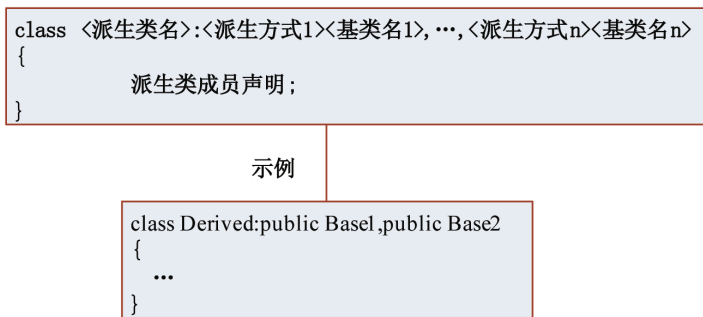


图 10-16 多重继承声明的一般形式

使用多重继承时应注意以下两点，如图 10-17 所示。

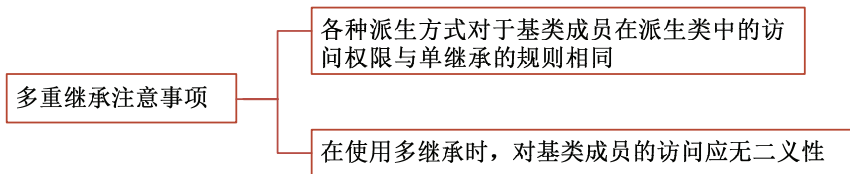


图 10-17 使用多重继承注意事项

### 10.4.2 二义性问题

在多重继承中，需要解决的主要问题是标识符不唯一，即二义性问题。例如，当在派生类



继承的多个基类中有同名成员时，派生类中就会出现标识符不唯一（二义性）的情况，这在程序中是不允许的。如图 10-18 所示的程序便存在二义性问题。

```
#include<iostream.h>
class Base1 → 定义基类Base1
{
public:
    int x;
    int a();
    int b();
};
class Base2 → 定义基类Base2
{
    int x;
    int a();
public:
    float b();
};
class Derived:Base1,Base2 → 类Derived私有继承自类Base1和Base2
{
};

void d(Derived &e) → 定义函数调用成员
{
    e.x=10;
    e.a();
    e.b();
}

int main()
{
    Derived ob;
    return 0;
}
```

错误，不知道x、a()、b()是从哪个基类继承来的，有二义性

图 10-18 程序中存在二义性问题

在多重继承中，派生类由多个基类派生时，基类之间用逗号隔开，且其每个基类前都必须指明继承方式，否则，默认为私有继承。可以通过 3 种方式来解决多重继承中的二义性问题，如图 10-19 所示。

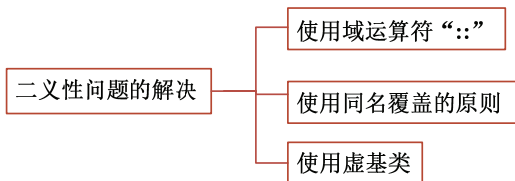


图 10-19 解决二义性问题的 3 种方式

关于虚基类的有关问题将在下一节讨论，这里先介绍前两种方法。

### 1. 使用域运算符

如果派生类的基类之间没有继承关系，同时又没有共同的基类，则在引用同名成员时，可在成员名前加上类名和域运算符来区别来自不同基类的成员。例如，将图 10-16 中的函数 d(Derived &e)()改写成如图 10-20 所示的形式。



```
void d(Derived &e)
{
    e.Base1::x=10;
    e.Base1::a();
    e.Base2::b();
}
```

图 10-20 使用域运算符

## 2. 使用同名覆盖的原则

在派生类中重新定义与基类中同名的成员（如果是成员函数，则参数表也要相同，参数不同为重载）以隐蔽掉基类中的同名成员。在引用这些同名的成员时，引用的就是派生类中的成员，这样二义性问题就得到了解决。

【示例 10-5】下列程序实现了同名覆盖，并且使用了域运算符，其实现代码及结果如图 10-21 所示。

```
#include <iostream h>
class Base
{
public:
    int x;
    void show()
    {
        cout<<"This is Base x= "<<x<<endl;
    };
};
class Derived:public Base
{
public:
    int x;

    void show()
    {
        cout<<"This is Derived x= "<<x<<endl;
    }
};
int main()
{
    Derived ob;
    ob.x=5;

    ob.show();

    ob.Base::x=12;

    ob.Base::show();
    return 0;
}
```

定义基类 Base

类 Derived 公有继承自类 Base

同名数据成员

同名成员函数

用同名覆盖原则引用派生类数据成员

用同名覆盖原则引用派生类成员 函数

用作用域运算符访问基类 数据成员

用作用域运算符访问基类成员 函数

图 10-21 使用同名覆盖原则、域运算符实例



【示例 10-6】下面的程序定义的派生类 `Derived` 有两个基类：`Base1` 和 `Base2`，其公有继承于这两个类，这就是多重继承。其实现代码及结果如图 10-22 所示。

```
#include <iostream h>
class Base1
{
public:
    int x;
    void print()
    {
        cout<<"class Base1: x= "<<x<<endl;
    }
};
class Base2
{
public:
    int x;
    void print()
    {
        cout<<"class Base2: x= "<<x<<endl;
    }
};
class Derived:public Base1,public Base2
{
public:
    int x;
    void print()
    {
        cout<<"class Derived: x= "<<x<<endl;
    }
};
int main()
{
    Derived ob;
    ob.x=10;
    ob.print();
    ob.Base1::x=20;
    ob.Base1::print();
    ob.Base2::x=30;
    ob.Base2::print();
    return 0;
}
```

定义基类 `Base1`

定义基类 `Base2`

多重公有继承 `Base1` 和 `Base2`

定义派生类 同名数据成员

定义派生类 同名成员函数

创建派生类对象

用同名覆盖原则 调用派生类成员

用作用域运算符 调用基类 `Base1` 成员

用作用域运算符 调用基类 `Base2` 成员

```
"C:\Program Files (x8...
class Derived: x= 10
class Base1: x= 20
class Base2: x= 30
Press any key to continue
```

图 10-22 多重继承实例



**提示：**使用作用域运算符来指明对象调用的某个成员属于哪个基类是很直观的方法，因此在多重继承中使用非常广泛。

### 10.4.3 多重继承的构造函数和析构函数

与单继承类似，使用多重继承时，也涉及基类成员、对象成员和派生类成员的初始化问题及内存释放和善后处理工作，因此，必要时也要定义构造函数和析构函数。在 C++ 中，声明多重继承构造函数的一般形式如图 10-23 所示。

```
<派生类名>::<派生类名>(参数总表)::基类名1(参数表1), ..., 基类名n(参数表n),  
对象成员名1(对象成员参数表1), ..., 对象成员名m(对象成员参数表m)  
{  
    派生类新增成员的初始化语句  
}
```

示例

```
Derived(int x, int y, int z, int v):Base1(x), Base2(y), ob1(z), ob2(v)  
{  
    ...  
}
```

图 10-23 声明多重继承构造函数的一般形式

多重继承构造函数和析构函数的执行顺序与单继承的相同，但需强调的是，基类之间的执行顺序是严格按照声明时从左到右的顺序来执行的，与它们在定义派生类构造函数中的次序无关。

【示例 10-7】下面的程序定义了多重继承的构造函数和析构函数，需仔细理解其执行顺序，其实现代码及结果如图 10-24 所示。



```
#include <iostream h>
class Base1 —————> 定义基类 Base1
{
    int x1;
    public:
        Base1(int y1) —————> 定义基类 Base1 的构造函数
        {
            x1=y1;
            cout<<"constructing Base1,x1="<<x1<<endl;
        }
        ~Base1() —————> 定义基类 Base1 的析构函数
        {
            cout<<"destructing Base1"<<endl;
        }
};
class Base2 —————> 定义基类 Base2
{
    int x2;
    public:
        Base2(int y2) —————> 定义基类 Base2 的构造函数
        {
            x2=y2;
            cout<<"constructing Base2,x2="<<x2<<endl;
        }
        ~Base2() —————> 定义基类 Base2 的析构函数
        {
            cout<<"destructing Base2"<<endl;
        }
};

class Derived:public Base2,public Base1 —————> 多重公有继承 Base1 和 Base2
{
    private:
        Base1 ob1;
        Base2 ob2; —————> 创建基类对象
    public:
        Derived(int x,int y,int z,int v):Base1(x),Base2(y),ob1(z),ob2(v)
        {
            cout<<"constructing Derived"<<endl;
        }
};
int main()
{
    Derived ob(1,2,3,4); —————> 创建派生类对象
    return 0;
}
```

派生类的构造函数

```
"C:\Program Files (x86)\Mi...
constructing Base2,x2=2
constructing Base1,x1=1
constructing Base1,x1=3
constructing Base2,x2=4
constructing Derived
destructing Base2
destructing Base1
destructing Base1
destructing Base2
Press any key to continue_
```

图 10-24 多重继承的构造函数和析构函数实例



## 10.5 虚基类

虚基类是指基类被 `virtual` 修饰, 用来解决基类中由于同名成员的问题而产生的二义性问题。前面讲解了解决二义性问题的两种方式, 本节将详细介绍另外一种方式——虚基类。

### 10.5.1 声明虚基类

虚基类的声明是在派生类的声明过程中进行的, 其声明的一般形式如图 10-25 所示。

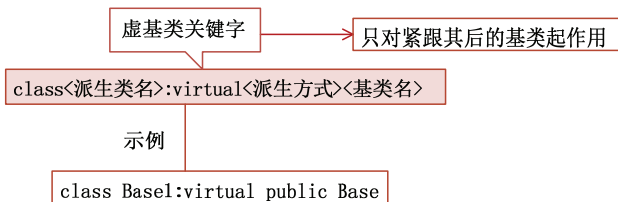


图 10-25 虚基类声明的一般形式

这种派生方式称为虚拟继承, 声明了虚基类以后, 虚基类的成员在进一步派生过程中和派生类一起维护同一个内存复制。

【示例 10-8】下面的程序定义类 `Base` 为虚基类, 仔细观察继承于虚基类的派生类的数据成员访问方式与前面学习的有何不同。其实现代码及结果如图 10-26 所示。

```
#include <iostream h>
class Base
{
protected:
    int x;
public:
    Base()
    {
        x=1;
    }
};

class Base1:virtual public Base
{
public:
    Base1()
    {
        cout<<"constructing Base1,x="<<x<<endl;
    }
};

class Base2:virtual public Base
{
public:
    Base2()
    {
        cout<<"constructing Base2,x="<<x<<endl;
    }
};

class Derived:public Base1,public Base2
{
public:
    Derived()
    {
        cout<<"constructing Derived x="<<x<<endl;
    }
};

int main()
{
    Derived obj;
    return 0;
}
```

定义基类 Base

定义 Base 为虚基类

定义派生类 Base1 的构造函数

定义 Base 为虚基类

定义派生类 Base2 的构造函数

定义多重继承于 Base1 和 Base2 的派生类

定义派生类 Derived 的构造函数

创建派生类对象

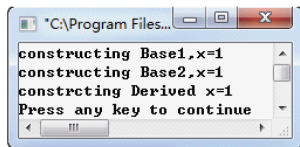


图 10-26 虚基类的实例



由于把公共基类 Base 声明为类 Base1 和类 Base2 的虚基类，所以由类 Base1 和类 Base2 派生的类 Derived 只有一个基类 Base，从而消除了二义性。

## 10.5.2 虚基类的构造函数和初始化

虚基类的初始化与一般多继承的初始化在语法上是一样的，但构造函数的执行顺序不同，如图 10-27 所示。

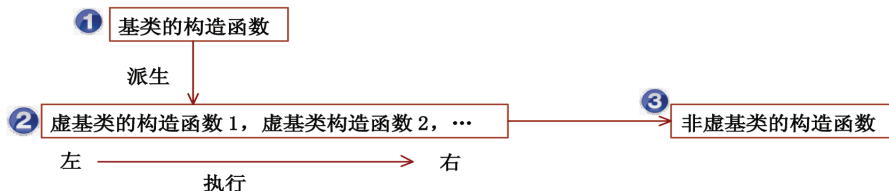


图 10-27 虚基类的构造函数的执行

【示例 10-9】下面的程序构建虚基类的构造函数，在创建该类对象的同时对其进行了初始化。仔细观察虚基类的构造函数执行顺序。其实现代码及结果如图 10-28 所示。

```
#include <iostream h>
class Base
{
protected:
    int x;
public:
    Base(int x1)
    {
        x=x1;
        cout<<"constructing Base,x="<<x<<endl;
    }
};
class Base1:virtual public Base
{
    int y;
public:
    Base1(int x1,int y1):Base(x1)
    {
        y=y1;
        cout<<"constructing Base1,y="<<y<<endl;
    }
};
class Base2:virtual public Base
{
    int z;
public:
    Base2(int x1,int z1):Base(x1)
    {
        z=z1;
        cout<<"constructing Base2,z="<<z<<endl;
    }
};
class Derived:public Base1,public Base2
{
    int xyz;
public:
    Derived(int x1,int y1,int z1,int xyz1):Base(x1),Base1(x1,y1),Base2(x1,z1)
    {
        xyz=xyz1;
        cout<<"constructing Derived xyz="<<xyz<<endl;
    }
};
int main()
{
    Derived obj(1,2,3,4);
    return 0;
}
```

定义基类 Base

定义基类 Base 的构造函数

定义基类 Base 为虚基类

定义 Base1 的构造函数

定义基类 Base 为虚基类

定义 Base2 的构造函数

定义多重继承的派生类

派生类构造函数

创建派生类对象

图 10-28 虚基类的构造函数和初始化实例





从如图 10-28 所示的结果图可以看出，虚基类 Base 的构造函数只执行了一次。这是因为当派生类 Derived 调用了虚基类 Base 的构造函数之后，类 Base1 和类 Bbase2 对虚基类 Base 构造函数的调用被忽略了。这是初始化虚基类和初始化非虚基类的不同。

在使用虚基类时应注意几个问题，如图 10-29 所示。

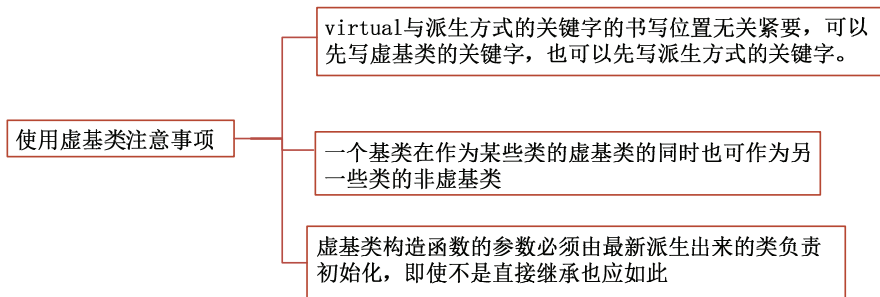


图 10-29 使用虚基类时注意的问题



## 10.6 友元

友元是用关键字 friend 修饰的，它提供了不同类或对象的成员函数之间、类的成员函数与一般函数之间进行数据共享的机制。本节将详细介绍友元的相关知识。

### 10.6.1 友元的引入

在 C++ 中，引入友元的目的如图 10-30 所示。



图 10-30 引入友元的目的

根据友元所指目标的不同，可分为友元函数和友元类，如图 10-31 所示。

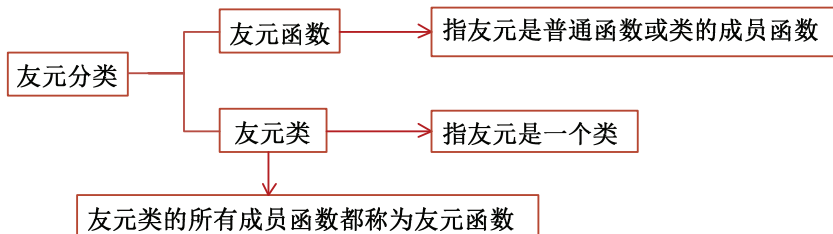


图 10-31 友元分类

### 10.6.2 友元函数

友元函数与普通成员函数不同，它不是当前类的成员函数，而是独立于当前类的外部函数；它可以是普通函数或其他类的成员函数。友元函数定义后可以访问该类的所有对象的所有成员，包括私有成员、保护成员和公有成员。



友元函数使用前必须要在类定义时声明；其定义既可以在类内部进行，也可以在类外部进行，但通常都定义在类的外部。C++将普通函数声明为友元函数的一般形式如图 10-32 所示。

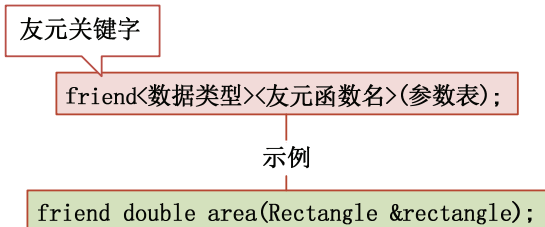


图 10-32 友元函数声明的一般形式

【示例 10-10】下面的程序将一个函数声明为友元函数，并在主程序中调用该函数对类的成员进行访问，实现代码及结果如图 10-33 所示。

```
#include <iostream.h>
class Rectangle
{
    double length, width;
public:
    Rectangle(double a=0,double b=0)
    {
        length =a;
        width =b;
    }
    Rectangle(Rectangle &r);
    friend double area(Rectangle&rectangle);
};
double area(Rectangle &rectangle)
{
    return (rectangle.length* rectangle.width);
}
int main()
{
    Rectangle ob(4, 5);
    cout<<"The area is:"<< area(ob)<<endl;
    return 0;
}
```

定义私有数据成员

定义构造函数

重载构造函数

声明友元函数

定义友元函数

创建对象

调用友元函数

图 10-33 友元函数实例

一般来说，使用友元函数应注意如图 10-34 所示的几个问题。

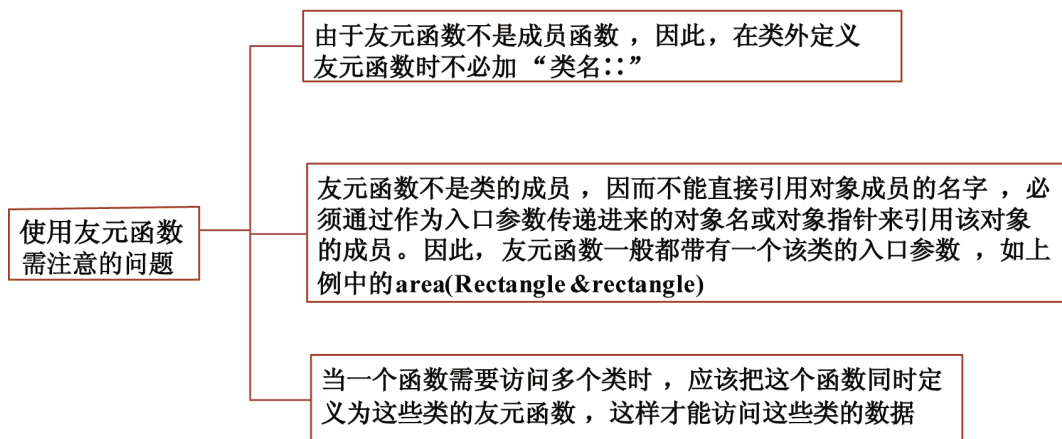


图 10-34 使用友元函数应注意的问题

### 10.6.3 友元成员

如果一个类的成员函数是另一个类的友元函数，则称这个成员函数为友元成员。通过友元成员可以访问的成员如图 10-35 所示。

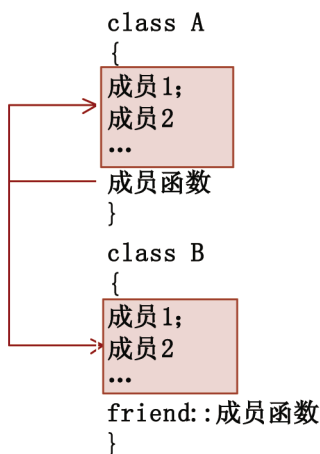


图 10-35 通过友元成员可以访问的成员

这种机制可使两个类相互访问，从而共同完成某些特定任务。

【示例 10-11】下面的程序声明一个类的成员函数是另一个类的友元函数，仔细观察其访问对象。其实现代码及结果如图 10-36 所示。



```
#include <iostream h>
#include <string>
class boy;
class girl
{
    char *name;
    int age;
public:
    girl(char *n, int a)
    {
        name=new char[strlen(n)+1];
        strcpy(name, n);
        age=a;
    }
    void prt(boy &b);
};
class boy
{
    char *name;
    int age;
public:
    boy(char *n, int a)
    {
        name=new char[strlen(n)+1];
        strcpy(name, n);
        age=a;
    }
    friend void girl::prt(boy &b);
};
void girl::prt (boy &b)
{
    cout<<"girl\'s name:"<<name<<"    age:"<<age<<"\n";
    cout<< boy\'s name: <<b.name<<"    age: <<b.age<<"\n";
}
int main()
{
    girl g1(" Stacy", 15);
    boy b1(" Jim", 16);
    g1.prt(b1);
    return 0;
}
```

声明类boy

定义类girl

定义构造函数

分配空间

调用字符串复制函数

声明公有成员函数

定义类 boy

定义构造函数

声明友元成员

定义友元成员

创建类girl的对象g1

创建类boy的对象b1

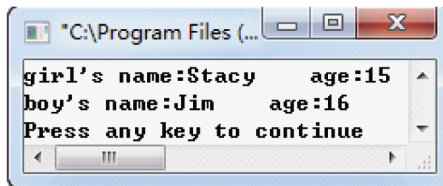


图 10-36 友元成员实例



在【示例 10-11】中，定义了 boy 和 girl 两个类，在类 boy 中声明了 girl 类的 prt() 成员函数为其友元成员，在类外定义了该成员函数。girl 类的 prt() 既可访问 girl 类的数据成员 name 和 age，也可访问 boy 类的数据成员 name 和 age。

以【示例 10-11】为例，使用友元成员时需注意，如图 10-37 所示的几个问题。

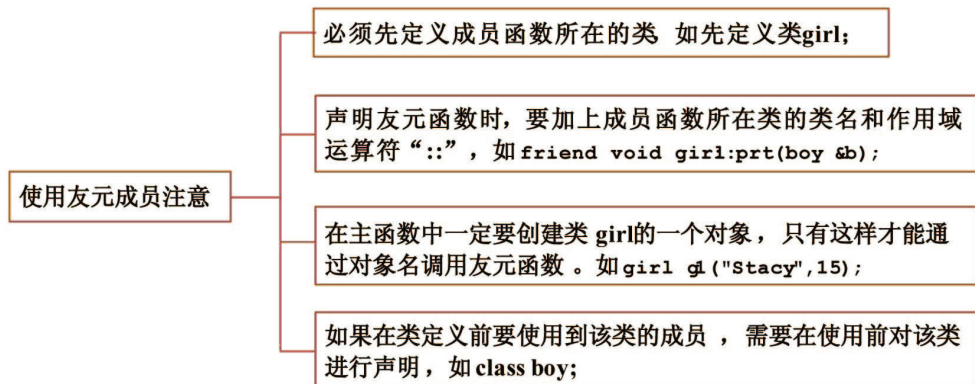


图 10-37 使用友元成员时需注意的问题

#### 10.6.4 友元类

如果一个类作为另一个类的友元，称这个类为友元类。当一个类成为另一个类的友元类时，这个类的所有成员函数都成为另一个类的友元函数。友元类中的所有成员函数都可以通过对象名直接访问另一个类中的所有成员，从而实现不同类之间的数据共享。

在 C++ 中，友元类声明的一般形式如图 10-38 所示。



图 10-38 友元类声明的一般形式

【示例 10-12】下面的程序声明两个类，其中一个类为另外一个类的友元，仔细观察其相互访问各自成员的权限。其实现代码及结果如图 10-39 所示。



```
#include <iostream h>
#include <string>
class boy;
class girl
{
    char *name;
    int age;
public:
    girl(char *n,int a)
    {
        name=new char[strlen(n)+1];
        strcpy(name, n);
        age=a;
    }
    void prt(boy &b);
};
class boy
{
    char *name;
    int age;
    friend class girl;
public:
    boy(char *n, int a)
    {
        name=new char[strlen(n)+1];
        strcpy(name, n);
        age=a;
    }
};
void girl::prt (boy &b)
{
    cout<<"girl\'s name:"<<name<<"    age:"<<age<<"\n";
    cout<<"boy\'s name:"<<b.name<<"    age:"<<b.age<<"\n";
}
int main()
{
    girl g1("Stacy",15);
    boy b1("Jim",16);
    g1.prt(b1);
    return 0;
}
```

声明类boy

定义类girl

定义构造函数

分配空间

调用字符串复制函数

声明公有成员函数

定义类 boy

声明类 girl 为类 boy 的友元类

定义构造函数

定义友元成员 函数

创建类 girl 的对象 g1

创建类 boy 的对象 b1

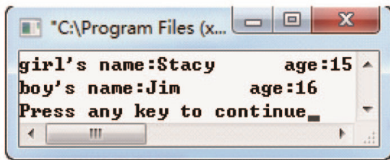


图 10-39 友元类实例



## 10.7 综合实例

本节将给出一个日期时间类的实例。帮助读者巩固类的继承和父类成员的使用方法等基础知识。



【示例 10-13】下面的程序首先定义了 CDate 和 CTime，然后派生出了 CDateTime。CDate 定义如图 10-40 所示，CTime 定义如图 10-41 所示，CDateTime 类如图 10-42 所示，主函数及运行结果如图 10-43 所示。

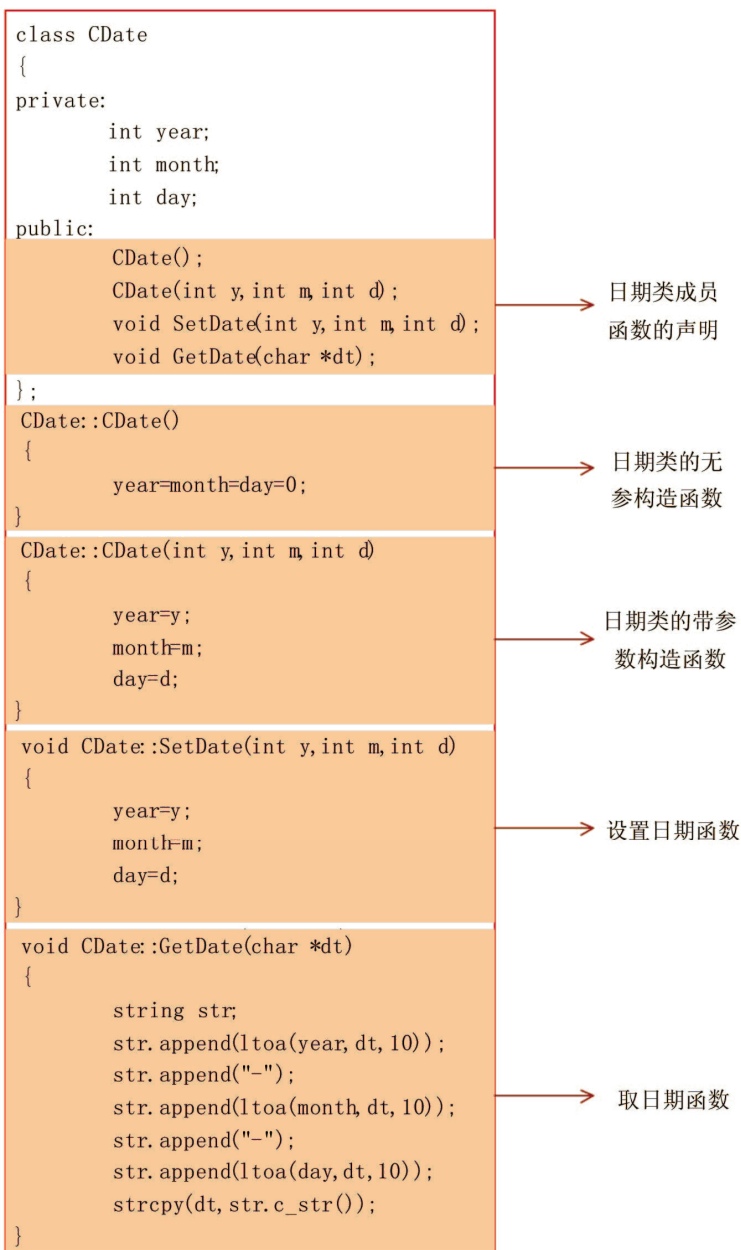


图 10-40 日期基类

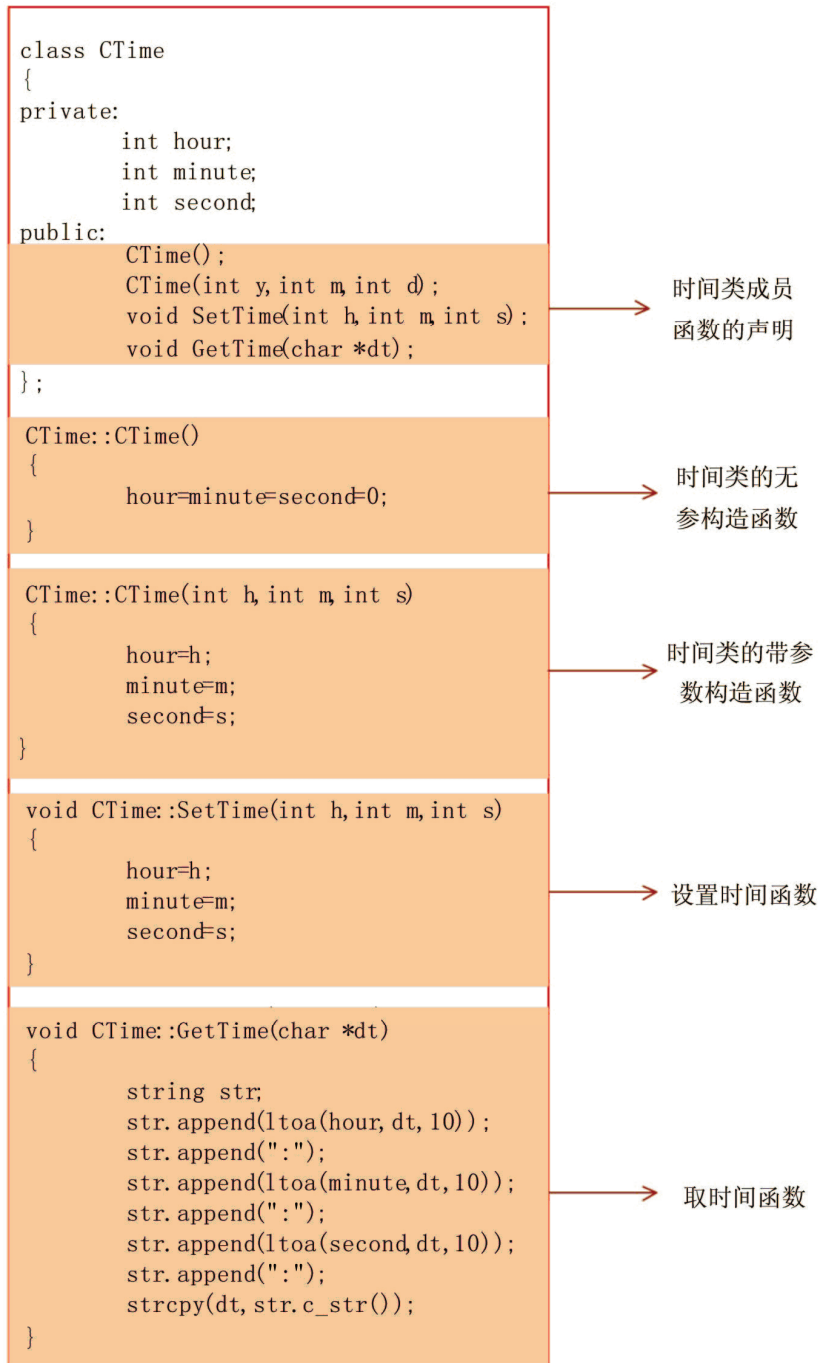


图 10-41 时间基类



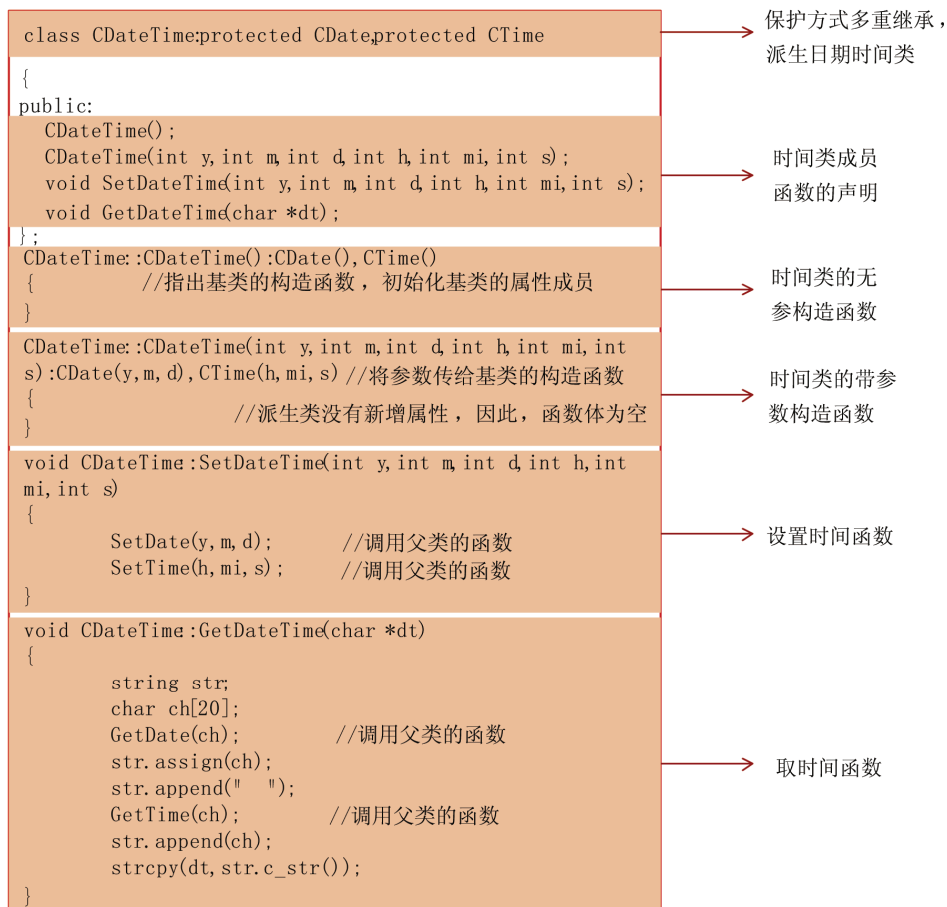


图 10-42 派生日期时间类

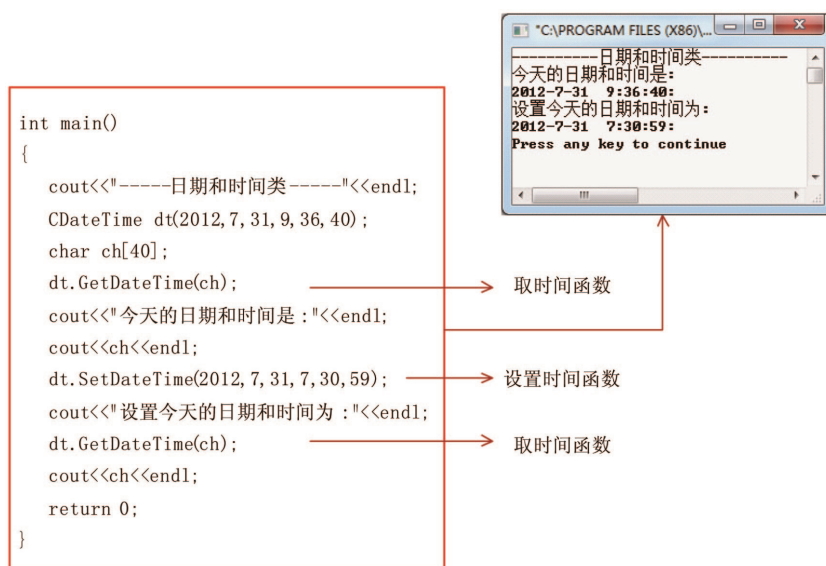


图 10-43 运行结果



## 10.8 小结

本章首先对继承和派生进行了概述，并介绍了成员的访问；然后重点讲述了继承的访问控制 and 多重继承；最后对虚基类和友元做了详细介绍。

## 10.9 习题

**【题目 10-1】**下面是一个定义了二维坐标点的类 `point`。要求：编写一个三维坐标系点的类，它从 `point` 类继承而来。其中，在三维点类中编写一个计算点到原点(0,0,0)的距离。程序的运行结果如图 10-44 所示。

```
class point
{
private:
    int xPos;
    int yPos;
public:
    point(int x=0,int y=0)
    {
        xPos=x;
        yPos=y;
    }
    void disp()
    {
        cout<<"("<<xPos<<","<<yPos<<")"<<endl;
    }
    int GetX()
    {
        return xPos;
    }
    int GetY()
    {
        return yPos;
    }
    double distance()
    {
        return sqrt(GetX()*GetX()+GetY()*GetY());
    }
};
```

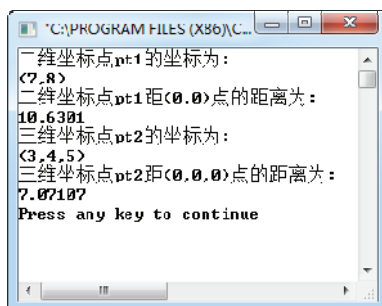


图 10-44 运行结果

**【题目分析】**本题要求读者熟悉类继承的相关知识，重点是掌握类继承的概念及作用。

**【关键代码】**

```
class point3D:public point
{
```



```
private:
    int zPos;
public:
    point3D(int x,int y,int z):point(x,y)
    {
        zPos=z;
    }
    void disp();
    double distance();
};
```

【题目 10-2】定义基类 BaseClass，并派生子类 ChildClass。分别将 BaseClass 的成员限制为 public、protected、private，设计不同的方法从 ChildClass 中访问父类成员。运行结果如图 10-45 所示。



图 10-45 运行结果

【题目分析】本题主要考查类的 3 种继承方式的区别。需注意的是，若本题中的 BaseClass 成员是私有的，则从子类中不能直接访问，但通过在 BaseClass 中定义非私有函数可以访问。

#### 【关键代码】

```
class BaseClass{
private:
    int x;
protected:
    int y;
public:
    int z;
protected:
    int GetX1() {
```



```
        return x;
    }
    int GetY1() {
        return y;
    }
    int GetZ1() {
        return z;
    }
public:
    BaseClass() {
        x=1;
        y=2;
        z=3;
    }
    int GetX2() {
        return x;
    }
    int GetY2() {
        return y;
    }
    int GetZ2() {
        return z;
    }
};

class ChildClass1:public BaseClass{
public:
    ChildClass1():BaseClass() {
    }
    void OutputX() {
        cout<<"在派生类中也不能直接访问 x. "<<endl;
        cout<<"通过调用基类的保护函数 GetX1 () 来访问 x: "<<GetX1 ()<<endl;
        cout<<"通过调用基类的公共函数 GetX2 () 来访问 x: "<<GetX2 ()<<endl;
    }
    void OutputY() {
        cout<<"在派生类中直接访问 y: "<<y<<endl;
        cout<<"通过调用基类的保护函数 GetY1 () 来访问 y: "<<GetY1 ()<<endl;
        cout<<"通过调用基类的公共函数 GetY2 () 来访问 y: "<<GetY2 ()<<endl;
    }
    void OutputZ() {
        cout<<"在派生类中直接访问 z: "<<z<<endl;
        cout<<"通过调用基类的保护函数 GetZ1 () 来访问 z: "<<GetZ1 ()<<endl;
        cout<<"通过调用基类的公共函数 GetZ2 () 来访问 z: "<<GetZ2 ()<<endl;
    }
};

class ChildClass2:private BaseClass{
public:
    ChildClass2():BaseClass() {
    }
    void OutputX() {
        cout<<"在派生类中也不能直接访问 x."<<endl;
        cout<<"通过调用基类的保护函数 GetX1 () 来访问 x: "<<GetX1 ()<<endl;
        cout<<"通过调用基类的公共函数 GetX2 () 来访问 x: "<<GetX2 ()<<endl;
    }
    void OutputY() {
        cout<<"在派生类中直接访问 y: "<<y<<endl;
        cout<<"通过调用基类的保护函数 GetY1 () 来访问 y: "<<GetY1 ()<<endl;
        cout<<"通过调用基类的公共函数 GetY2 () 来访问 y: "<<GetY2 ()<<endl;
    }
    void OutputZ() {
        cout<<"在派生类中直接访问 z: "<<z<<endl;
    }
};
```



```

        cout<<"通过调用基类的保护函数 GetZ1 () 来访问 z: "<<GetZ1 ()<<endl;
        cout<<"通过调用基类的公共函数 GetZ2 () 来访问 z: "<<GetZ2 ()<<endl;
    }
};

class ChildClass3:protected BaseClass{
public:
    ChildClass3():BaseClass() {
    }
    void OutputX() {
        cout<<"在派生类中也不能直接访问 x."<<endl;
        cout<<"通过调用基类的保护函数 GetX1 () 来访问 x: "<<GetX1 ()<<endl;
        cout<<"通过调用基类的公共函数 GetX2 () 来访问 x: "<<GetX2 ()<<endl;
    }
    void OutputY() {
        cout<<"在派生类中直接访问 y: "<<y<<endl;
        cout<<"通过调用基类的保护函数 GetY1 () 来访问 y: "<<GetY1 ()<<endl;
        cout<<"通过调用基类的公共函数 GetY2 () 来访问 y: "<<GetY2 ()<<endl;
    }
    void OutputZ() {
        cout<<"在派生类中直接访问 z: "<<z<<endl;
        cout<<"通过调用基类的保护函数 GetZ1 () 来访问 z: "<<GetZ1 ()<<endl;
        cout<<"通过调用基类的公共函数 GetZ2 () 来访问 z: "<<GetZ2 ()<<endl;
    }
};

cout<<"公共继承: "<<endl;
ChildClass1 c1;
cout<<"子类对象不能直接使用 x."<<endl;
c1.OutputX();
cout<<"子类对象不能直接使用 y."<<endl;
c1.OutputY();
cout<<"子类对象可以直接使用 z:"<<c1.z<<endl;
c1.OutputZ();
cout<<endl;
cout<<"私有继承: "<<endl;
ChildClass2 c2;
cout<<"子类对象不能直接使用 x."<<endl;
c2.OutputX();
cout<<"子类对象不能直接使用 y."<<endl;
c2.OutputY();
cout<<"子类对象不能直接使用 z."<<endl;
c2.OutputZ();
cout<<endl;
cout<<"保护继承: "<<endl;
ChildClass3 c3;
cout<<"子类对象不能直接使用 x."<<endl;
c3.OutputX();
cout<<"子类对象不能直接使用 y."<<endl;
c3.OutputY();
cout<<"子类对象不能直接使用 z."<<endl;
c3.OutputZ();

```

**【题目 10-3】**鸭嘴兽是一种很奇怪的动物，如图 10-46 示。鸭嘴兽兼具有哺乳动物和鸟类的特征。鸭嘴兽具有和鸭子一样的嘴巴，更关键的是它靠下蛋来繁殖后代，这明显是鸟类的特征。然而，鸭嘴兽又靠乳汁来哺育幼仔，这又是哺乳动物的重要特征。下面请读者定义一个鸭嘴兽类，既具有鸟类卵生的特性，又具有哺乳动物类用乳汁喂养后代的特性。程序运行结果如图 10-47 示。

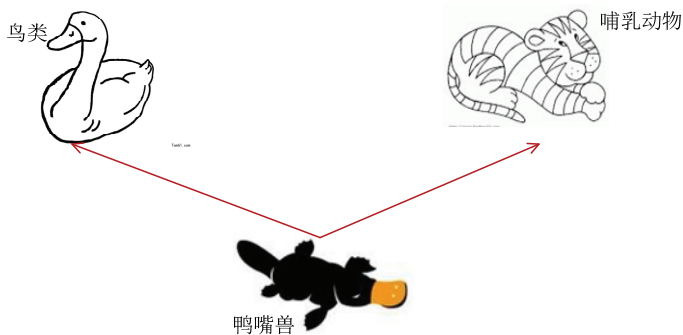


图 10-46 鸭嘴兽

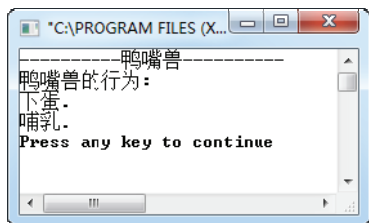


图 10-47 运行结果

【题目分析】 本题考查多继承的基本知识。

【关键代码】

```
class Bird
{
public:
    void LayEggs()
    {
        cout<<"下蛋."<<endl;
    }
};

class Mammal
{
public:
    void Suckle()
    {
        cout<<"哺乳."<<endl;
    }
};

class DuckBill:public Bird,
               public Mammal
{
};
```

【题目 10-4】水陆两栖坦克作战能力很强，可以跋山涉水。水陆两栖坦克既具有坦克的特点，也有船的特性。而且坦克和船都是运输工具。编号是运输工具都应该有的属性，所以应当放在工具类中。为了避免多继承导致的数据重复，在由运输工具类派生坦克类和轮船类时应当使用虚拟继承。请读者设计一个水陆两栖坦克的类。程序运行的结果如图 10-48 所示，水陆两栖坦克的类层次结构如图 10-49 所示。

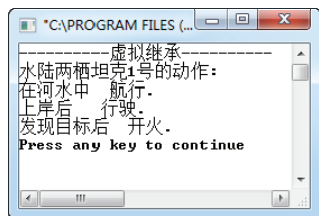


图 10-48 运行结果

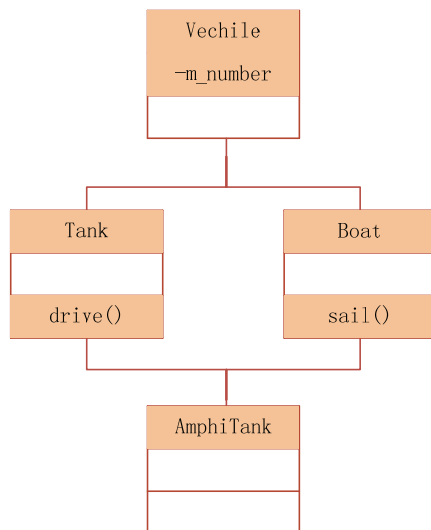


图 10-49 水陆两栖坦克的类层次结构

【题目分析】本题主要考查虚拟继承的使用。

【关键代码】

```

class Vehicle
{
public:
    int m_number;
};

class Tank:public virtual Vehicle
{
public:
    void drive()
    {
        cout<<"行驶."<<endl;
    }
    void fire()
    {
        cout<<"开火."<<endl;
    }
};

class Boat:public virtual Vehicle
{
public:
    void sail()
    {
        cout<<"航行."<<endl;
    }
};

class AmphiTank:public Tank,
                public Boat
{
};
  
```

【题目 10-5】在二维平面内定义一个点类 point，定义类 cz，cz 中可以修改 point 类定义的点的坐标，计算两点间的距离。cz 声明为类 point 的友元类。程序运行结果如图 10-50 所示。

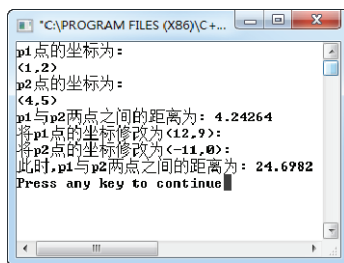


图 10-50 运行结果

【题目分析】 本题考查友元类的使用。

【关键代码】

```
class cz;
class point
{
private:
    int x;
    int y;
    friend cz;
public:
    point(int i=0,int j=0)
    {
        x=i;
        y=j;
    }
    void disp()
    {
        cout<<"("<<x<<"", "<<y<<"")"<<endl;
    }
};

class cz
{
public:
    double dis(point &p1,point &p2)
    {
        double d;
        d=sqrt((p1.x-p2.x)*(p1.x-p2.x)+(p1.y-p2.y)*(p1.y-p2.y));
        return d;
    }
    void Set(point *p1,int a,int b)
    {
        p1->x=a;
        p1->y=b;
    }
};

cz cz1;
point p1(1,2);
point p2(4,5);
cout<<"p1 点的坐标为: "<<endl;
p1.disp();
cout<<"p2 点的坐标为: "<<endl;
p2.disp();
cout<<"p1 与 p2 两点之间的距离为: "<<cz1.dis(p1,p2)<<endl;
cout<<"将 p1 点的坐标修改为(12,9): "<<endl;
cz1.Set(&p1,12,9);
cout<<"将 p2 点的坐标修改为(-11,0): "<<endl;
cz1.Set(&p2,-11,0);
cout<<"此时,p1 与 p2 两点之间的距离为: "<<cz1.dis(p1,p2)<<endl;
```



# 第 11 章 多态

多态是面向对象开发过程中一个非常重要的概念，可以说是面向对象的“核心”。只有体现出多态的特征，一个程序才能称得上是面向对象的程序。本章将详细介绍多态的有关知识。



## 11.1 多态概述

### 11.1.1 什么是多态

多态 (polymorphism)，从字面理解是“多种形态，多种形式”，是一种将不同的特殊行为泛化为单个特殊记号的机制。在面向对象中，多态的描述如图 11-1 所示。

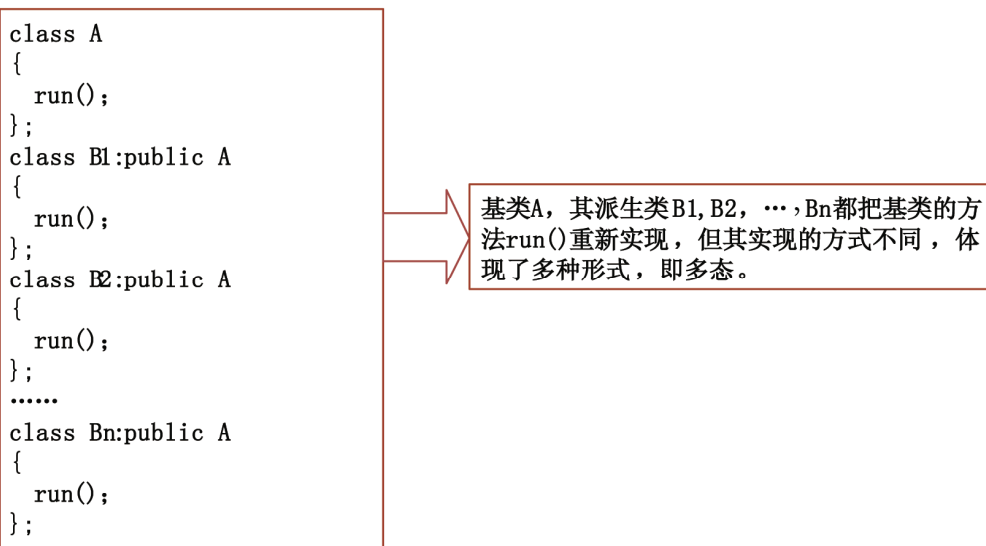


图 11-1 多态的描述

多态从实现的角度可划分为两类，如图 11-2 所示。

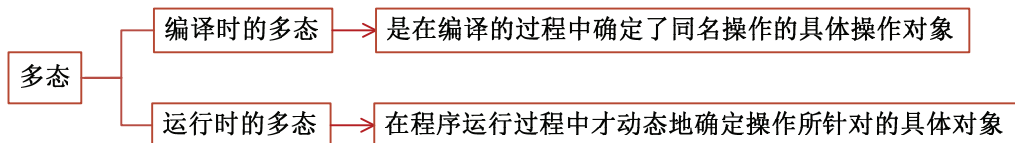


图 11-2 多态从实现的角度划分



多态也是人类思维方式的一种直接模拟。例如，图 11-3 表示的是求两个数最大值的多种表示，其实这些表示都可以通过一个统一的标识来表示。

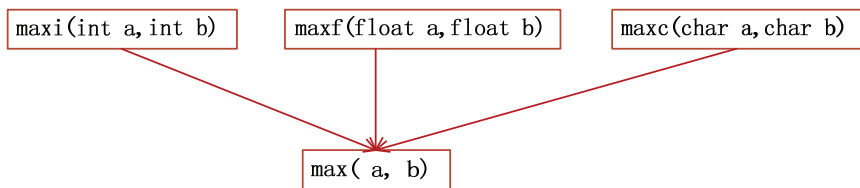


图 11-3 多态的含义

### 11.1.2 多态的引入

为了让读者更好地理解多态在实际程序中的应用，下面给出一个实例，引出要使用多态的原因。其实现代码如图 11-4 所示。

```
#include <iostream.h>
class Animal
{
public:
    void sleep()
    {
        cout<<"Animal sleep"<<endl;
    }
    void breathe()
    {
        cout<<"Animal breathe"<<endl;
    }
};
class Fish:public Animal
{
public:
    void breathe()
    {
        cout<<"Fish bubble"<<endl;
    }
};
int main()
{
    Fish fh;
    Animal *an=&fh;
    an->breathe();
    return 0;
}
```

Annotations for Figure 11-4:

- 定义基类 Animal (points to `class Animal`)
- 定义成员函数 (points to `void sleep()`)
- 定义 Animal 的公有派生类 Fish (points to `class Fish:public Animal`)
- 定义同名成员函数 (points to `void breathe()` in the Fish class)
- 创建对象 (points to `Fish fh;`)
- 定义对象指针指向对象 fh (points to `Animal *an=&fh;`)
- 调用 breathe() 函数 (points to `an->breathe();`)

图 11-4 多态引入原因实例

在如图 11-4 所示的代码中，主函数 `main()` 中首先定义了一个 `Fish` 类的对象 `fh`，接着定义了一个指向 `Animal` 类的指针变量 `an`，将 `fh` 的地址赋给了指针变量 `an`，然后利用该变量调用 `an->breathe()`。读者往往将这种情况和 C++ 的多态性混淆，认为 `fh` 实际上是 `Fish` 类的对象，应该是调用 `Fish` 类的 `breathe()`，然后输出“Fish bubble”，但结果却不是这样的。具体分析过程如图 11-5 所示。

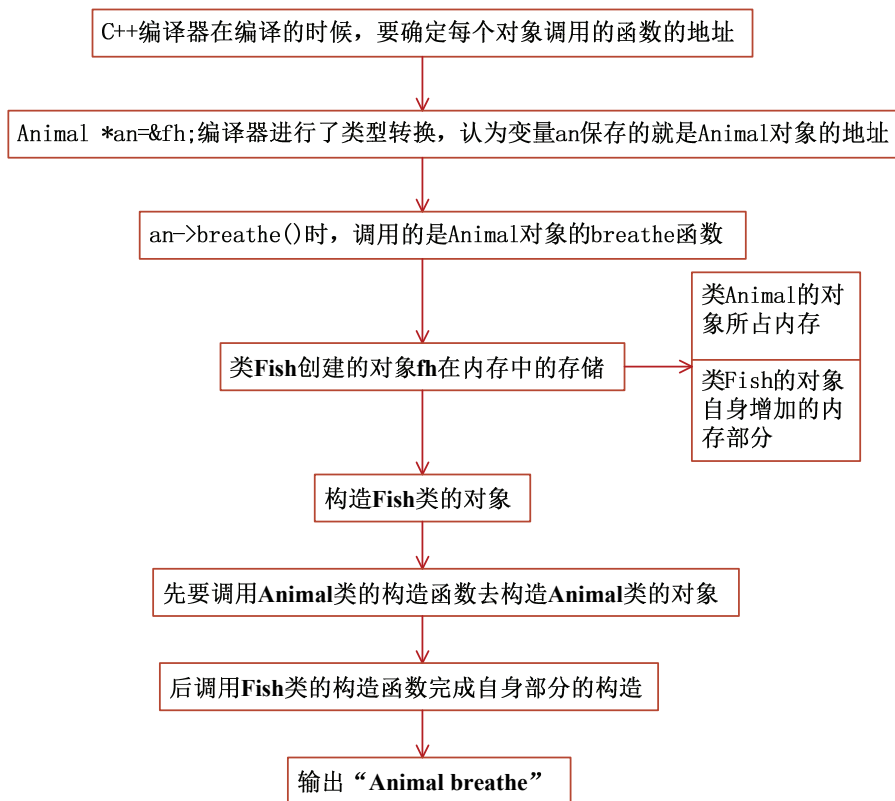


图 11-5 结果分析过程

图 11-4 中代码的运行结果并不是读者所希望看到的输出“Fish bubble”信息，这个问题就可以通过多态来解决。

### 11.1.3 联编

联编是确定操作的具体对象的过程，指计算机程序自身彼此关联的过程，即把一个标识符名和一个存储地址联系在一起的过程，如图 11-6 所示。

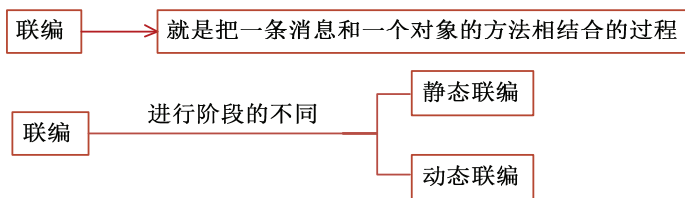


图 116 联编

这两种联编过程分别对应着多态的两种实现方式，如图 11-7 所示。

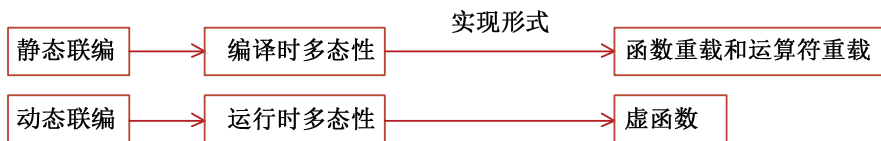


图 11-7 多态的两种实现方式



## 11.2 函数重载

在 C++ 中，可以用函数重载和运算符重载来实现编译时的多态性。本节只介绍函数重载，运算符重载将在第 12 章中讲解。

函数的重载也称多态函数，其具体作用及含义如图 11-8 所示。

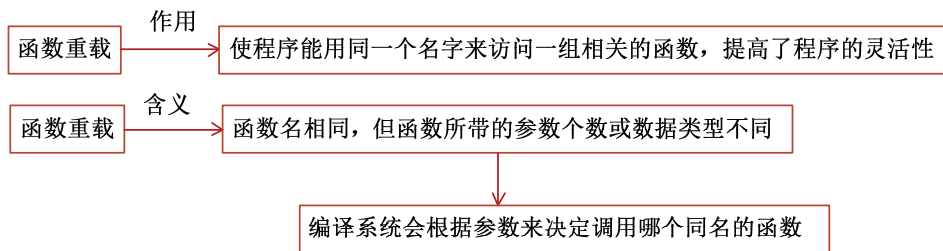


图 11-8 函数重载的作用及含义

面向对象程序设计中，函数的重载表现为两种情况，如图 11-9 所示。

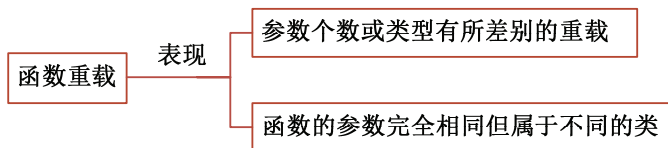


图 11-9 函数的重载表现为两种情况

关于图 11-9 中的第一种情况已在前面构造函数的重载中做了详细介绍，此处主要介绍第二种情况。当函数的参数完全相同但属于不同的类时，为了让编译能正确区分调用哪个类的同名函数，可采用两种方法，如图 11-10 所示。

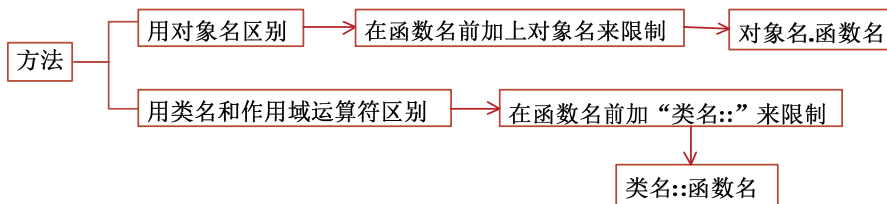


图 11-10 编译能正确区分调用哪个类的同名函数的方法

【示例 11-1】下面的程序分别采用对象名区别和类名加上运算符区别来实现成员函数的重载，其实现代码及结果如图 11-11 所示。

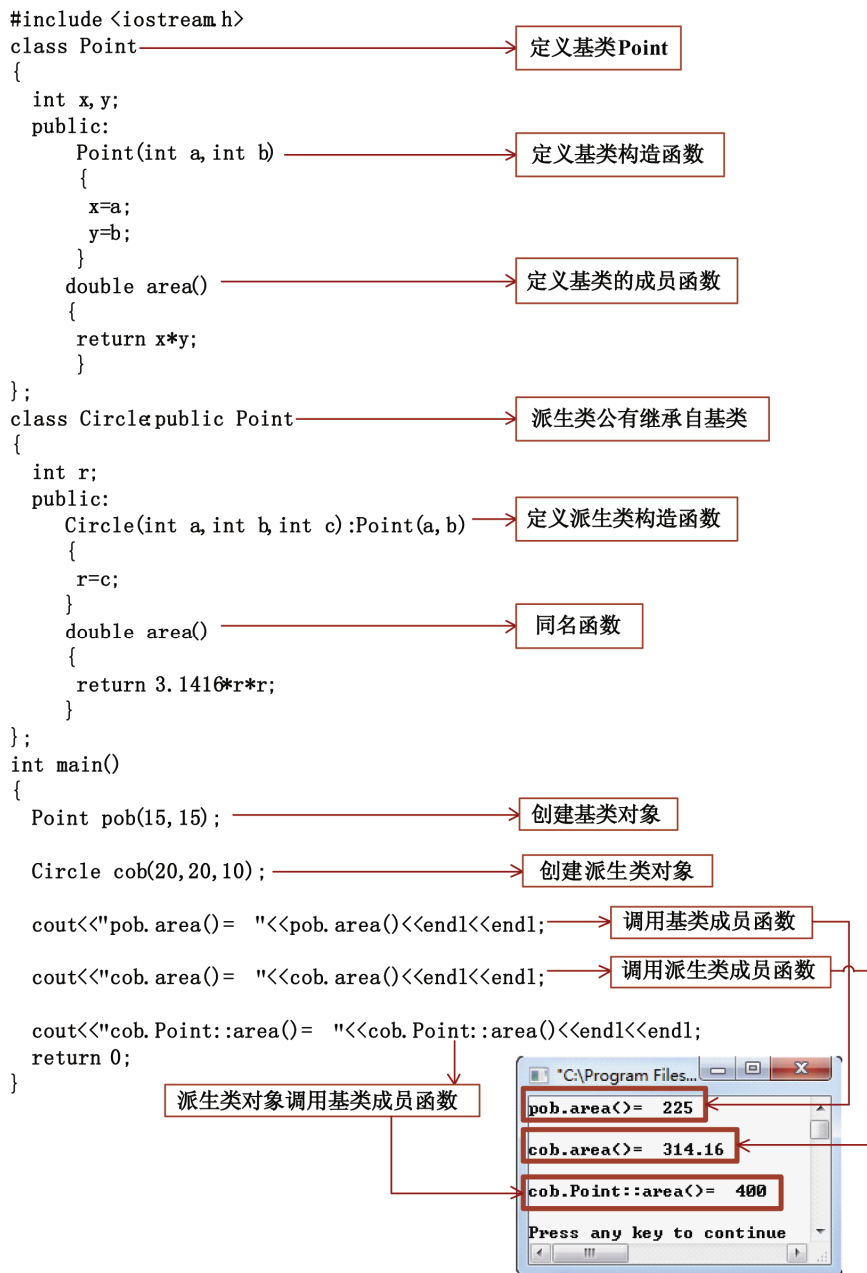


图 11-11 函数重载实例

通过函数重载方式，可以使同名函数实现不同的功能，即实现了静态多态性。



## 11.3 虚函数

在 C++ 中，虚函数是实现运行时多态的一个重要方式，是重载的另一种形式，实现的是动态的重载，即函数调用与函数体之间的联系是在运行时建立的，也就是动态联编。



### 11.3.1 定义虚函数

虚函数的定义是在基类中进行的，即把基类中需要定义为虚函数的成员函数声明为 virtual。虚函数定义的一般形式如图 11-12 所示。

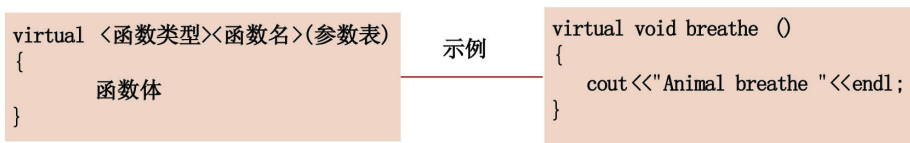


图 11-12 虚函数定义的一般形式

当基类中的某个成员函数被声明为虚函数后，就可以在派生类中重新定义。在派生类中重新定义时，其函数原型包括返回类型、函数名、参数个数和类型，参数的顺序都必须与基类中的原型完全一致。

【示例 11-2】下面的程序在基类中定义一个虚函数，当派生类公有继承于基类时，对该虚函数进行重定义，其实现代码及结果如图 11-13 所示。

```
#include <iostream.h>
class Animal
{
public:
    void sleep()
    {
        cout<<"Animal sleep"<<endl;
    }
    virtual void breathe()
    {
        cout<<"Animal breathe"<<endl;
    }
};
class Fish:public Animal
{
public:
    void breathe()
    {
        cout<<"Fish bubble"<<endl;
    }
};
int main()
{
    Fish fh;
    Animal *an=&fh;
    an->breathe();
    return 0;
}
```

定义基类 Animal

定义成员函数

定义虚函数

定义 Animal 的公有派生类 Fish

定义同名成员函数

创建对象

定义对象指针指向对象 fh

调用 breathe() 函数

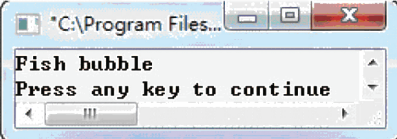


图 11-13 虚函数实例



在如图 11-13 所示的代码中，将基类中的成员函数 `breathe()` 定义为虚函数，即加上 `virtual` 关键字，然后在主函数 `main()` 中定义 `Animal` 对象指针指向类 `fish` 的对象 `fh`，调用 `breathe()` 函数后，得到的结果就是预期输出 `Fish bubble` 的结果。其具体分析过程如图 11-14 所示。

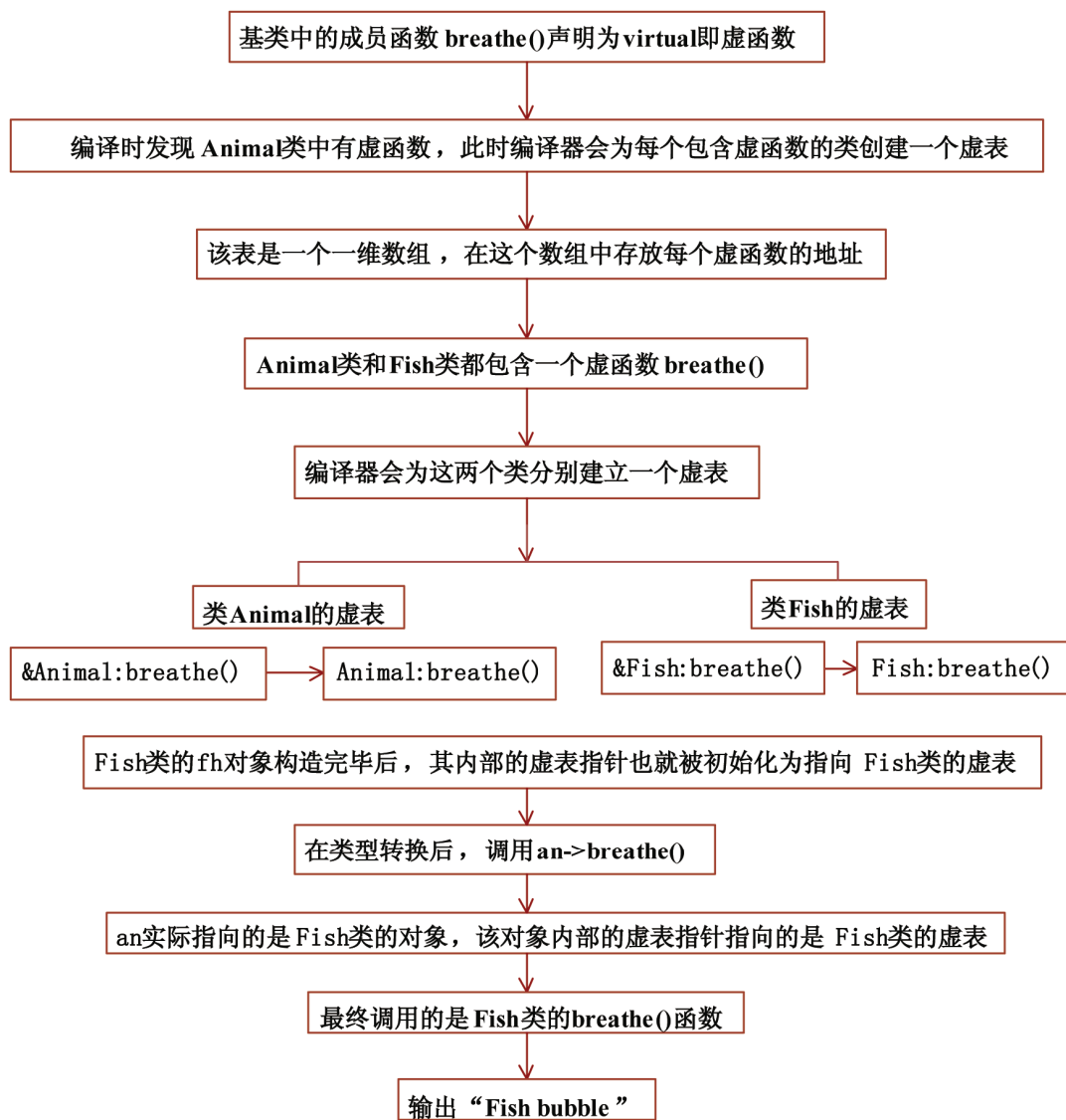


图 11-14 虚函数实例分析过程

在使用派生类对象指针时应注意的问题如图 11-15 所示。

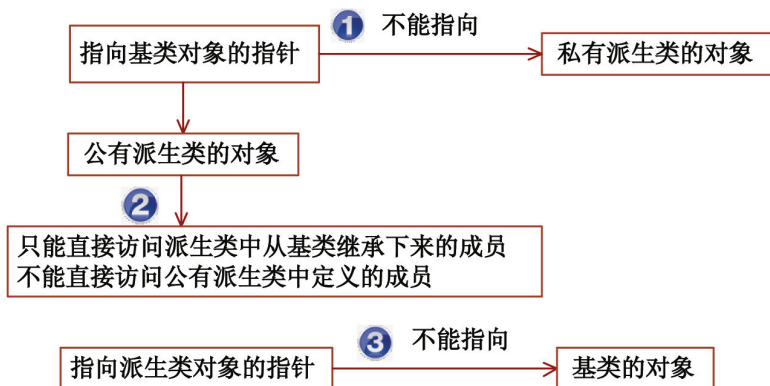


图 11-15 使用派生类对象指针时应注意的问题

虚函数可以很好地实现多态，在使用虚函数时应注意的问题如图 11-16 所示。

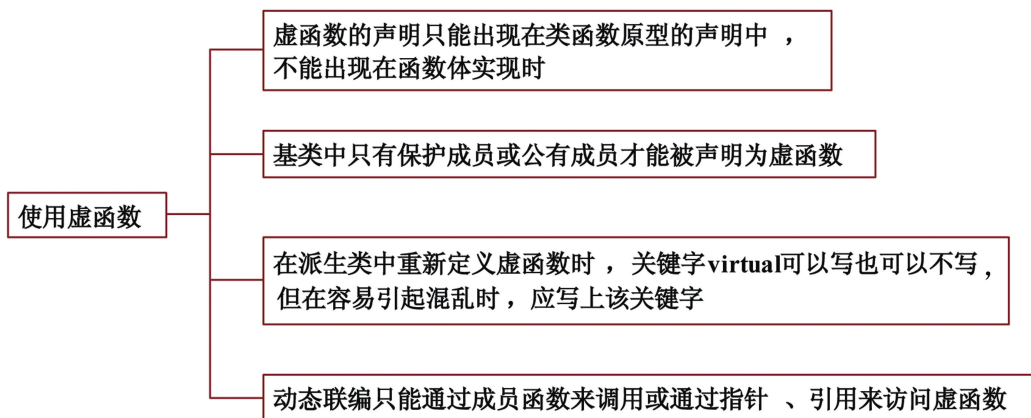


图 11-16 使用虚函数时应注意的问题

在派生类中重新定义基类中的虚函数，是函数重载的另一种形式，但它与函数重载又有区别，如图 11-17 所示。

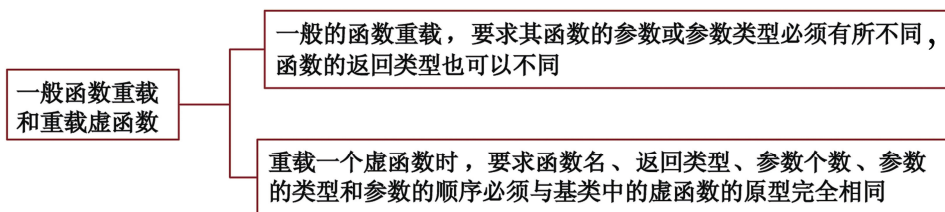


图 11-17 函数重载与重载虚函数的区别

### 11.3.2 多级继承和虚函数

多级继承可以看做是多个单继承的组合，多级继承的虚函数与单继承的虚函数的调用相同。即不同类创建的对象调用的函数是不一样的。

【示例 11-3】下面的程序实现了多级继承下的虚函数调用，其实现代码及结果如图 11-18 所示。



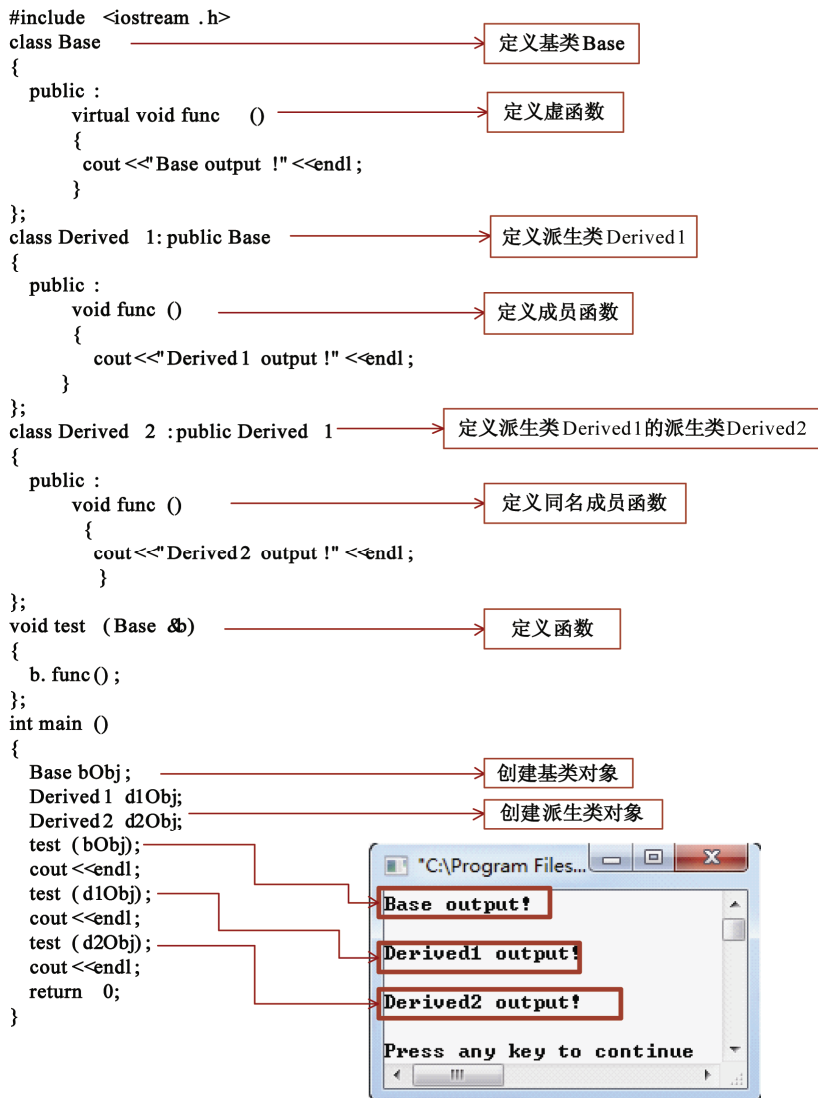


图 11-18 多级继承和虚函数实例

上述代码中定义了一个多级继承，在基类中定义了虚函数 `func()`，在主函数 `main()` 中调用该函数时，不同类创建的对象调用的函数是不一样的，即实现了多态的“一个接口，多种实现”的功能。



## 11.4 纯虚函数与抽象类

抽象类是一种包含纯虚函数的特殊的类。建立抽象类是为了多态地使用抽象类的成员函数。本节将介绍纯虚函数和抽象类。

### 11.4.1 纯虚函数

当在基类中不能为虚函数给出一个有意义的实现时，可以将其声明为纯虚函数。纯虚函数



的实现可以留给派生类来完成。纯虚函数的作用是为派生类提供一个一致的接口。一般来说，一个抽象类带有至少一个纯虚函数。纯虚函数定义的一般形式如图 11-19 所示。

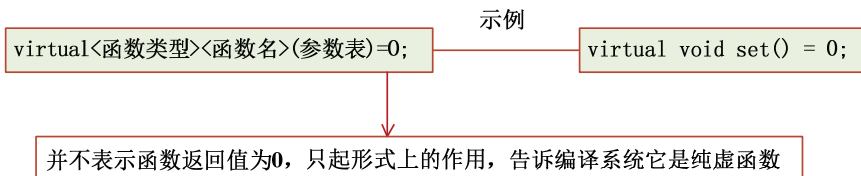


图 11-19 纯虚函数定义的一般形式

纯虚函数与普通虚函数定义的不同在于书写形式上加了“=0”，说明在基类中不用定义该函数的函数体。

【示例 11-4】下面的程序定义了纯虚函数，纯虚函数的函数体由派生类定义。其实现代码及结果如图 11-20 所示。

```
#include <iostream.h>
class Point
{
protected:
    int x0, y0;
public:
    Point(int i=0, int j=0)
    {
        x0=i;
        y0=j;
    }
    virtual void set() = 0;
};
class Line : public Point
{
protected:
    int x1, y1;
public:
    Line(int i=0, int j=0, int m=0, int n=0):Point(i, j)
    {
        x1=m;
        y1=n;
    }
    void set()
    {
        cout<<"Line::set() called \n";
    }
};
void setobj(Point *p)
{
    p->set();
}
int main()
{
    Line *lineobj = new Line;
    setobj(lineobj);
    return 0;
}
```

定义基类 Point

定义构造函数

声明纯虚函数

定义派生类 Line

定义构造函数

定义成员函数

定义虚函数

创建对象指针

调用虚函数

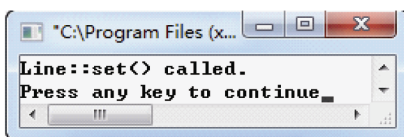


图 11-20 纯虚函数实例



11.4.2 抽象类

抽象类是包含纯虚函数的一种特殊的类，是为了抽象和设计而建立的，处于继承层次结构的较上层。抽象类是不能创建对象的，为了强调一个类是抽象类，可将该类的构造函数声明为保护的访问控制权限。抽象类的主要作用如图 11-21 所示。

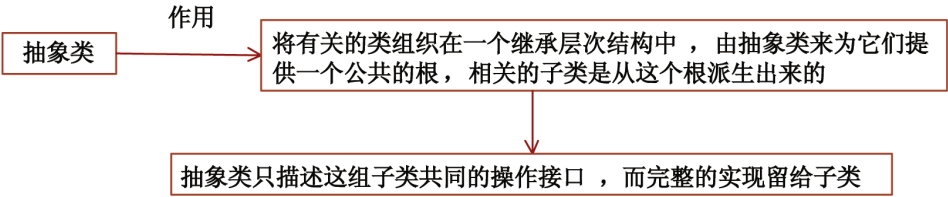


图 11-21 抽象类的主要作用

使用抽象类时应注意的问题如图 11-22 所示。

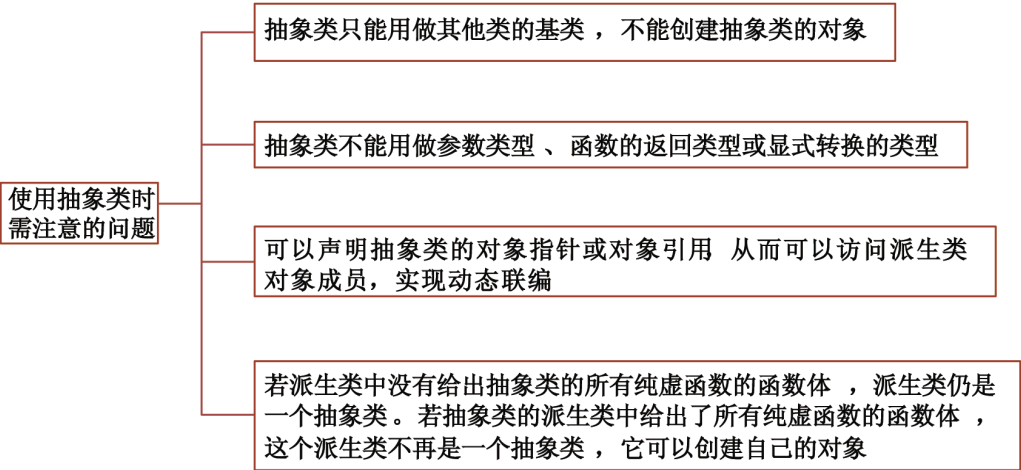


图 11-22 使用抽象类时应注意的问题

【示例 11-5】下面的程序定义一个抽象类，该抽象类中包含纯虚函数，同时定义公有继承于该抽象类的派生类，其实现代码及结果如图 11-23 所示。



```
#include <iostream.h>
class Vehicle
{
    protected:
        float speed;
        int total;
    public:
        Vehicle(float speed, int total)
        {
            Vehicle::speed = speed;
            Vehicle::total = total;
        }
        virtual void ShowMember()=0;
};
class Car:public Vehicle
{
    protected:
        int aird;
    public:
        Car(int aird, float speed, int total):Vehicle(speed, total)
        {
            Car::aird = aird;
        }
        void ShowMember()
        {
            cout<<"the speed is : "<<speed<<endl;
            cout<<"the total is : "<<total<<endl;
            cout<<"the aird is : "<<aird<<endl;
        }
};
int main()
{
    //Vehicle a(100,4);
    Car b(250, 150, 4);
    b.ShowMember();
    return 0;
}
```

定义抽象类 Vehicle

定义构造函数

声明纯虚函数

定义派生类

定义构造函数

派生类成员函数重载

错误，抽象类不能创建对象

创建对象

调用成员函数

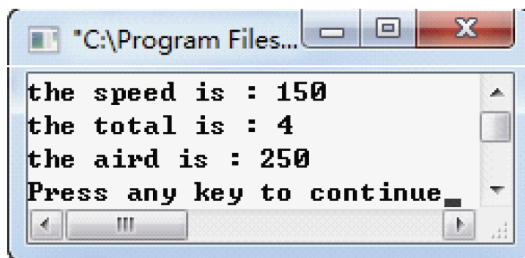


图 11-23 抽象类实例



## 11.5 综合实例

本节将通过一个综合实例，帮助读者深入学习多态、虚函数、纯虚函数、抽象类的知识。

【示例 11-6】图形类接口。对于一个图形，通常的操作有绘制、获取尺寸、变换位置和形状等，而这些操作都可以看做是图形类能够支持的接口。本例通过接口访问和操作具体的图形类。其中接口类（绘图接口和尺寸接口）、图形类、圆类的定义代码如图 11-24 所示，主函数及程序的运行结果如图 11-25 所示。

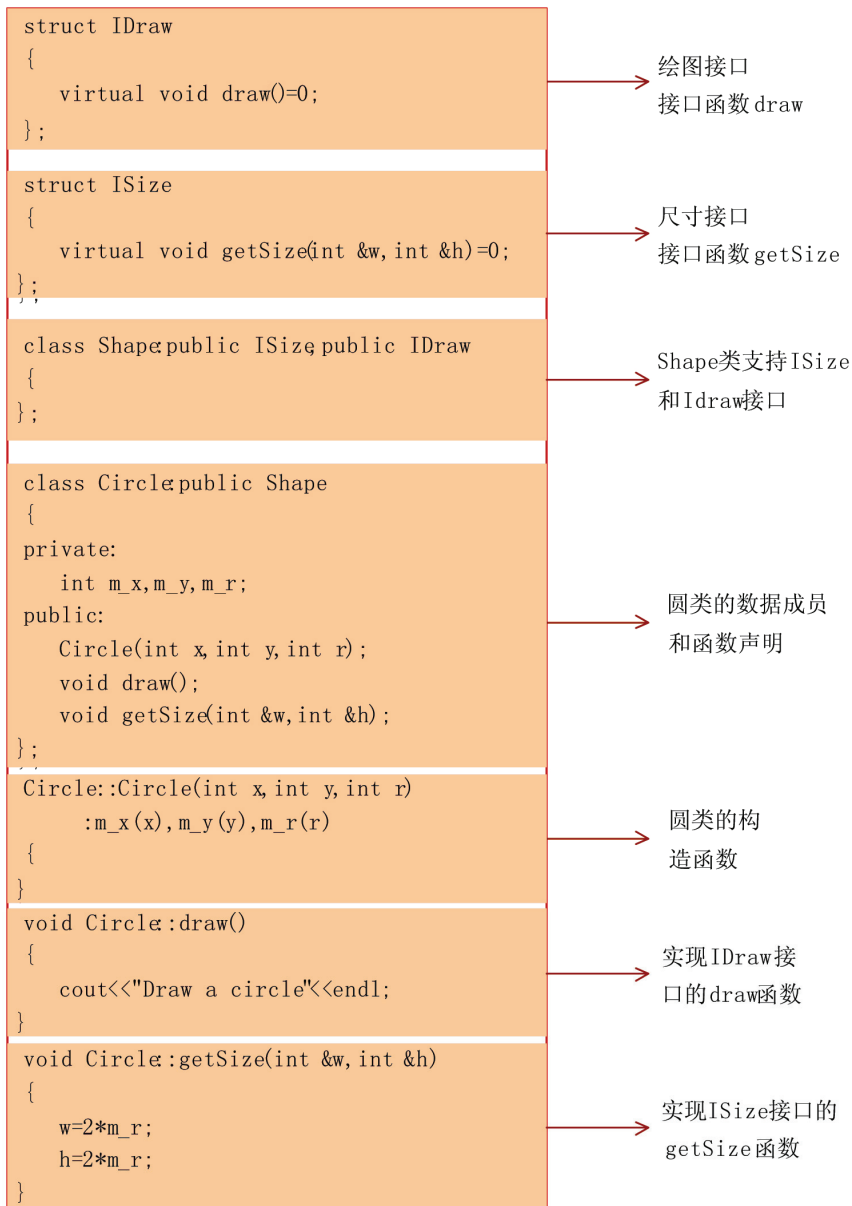


图 11-24 类代码



```
int main()
{
    Circle circle(0, 0, 123);
    IDraw *pDraw=&circle;           //获取IDraw接口指针
    pDraw->draw();                   //通过IDraw接口画圆
    ISize *pSize=&circle;           //获取ISize接口指针
    int w=0, h=0;
    pSize->getSize(w, h);            //通过ISize接口获取圆的高度
    cout<<"Width: "<<w<<endl;      //输出宽
    cout<<"Height: "<<h<<endl;     //输出高
    return 0;
}
```

运行结果

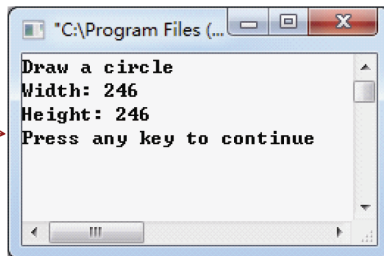


图 11-25 画图并输出圆的宽和高



## 11.6 小结

本章主要介绍了 C++面向对象程序设计的一个重要特征——多态及实现多态的两种形式：函数重载和虚函数。另外，对纯虚函数和抽象类也进行了详细介绍。



## 11.7 习题

【题目 11-1】一个公司在计算薪资时，会根据其职务采取不同的计算方法：经理的薪资可能是固定的，工程师的薪资则除了基本工资外，还可能会有项目奖金；而销售人员一般还会有提成。由此可见，职员的薪资计算方法是一种多态行为。请读者利用动态多态的机制（虚函数），计算不同职务职员的薪水。

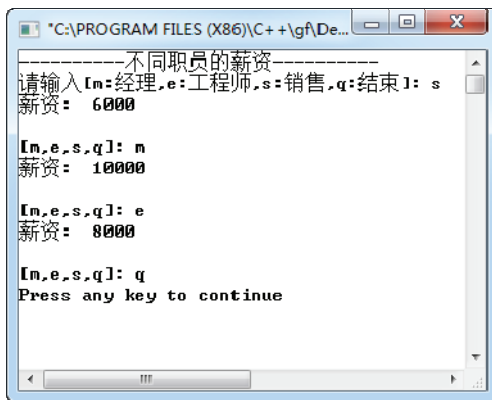


图 11-24 运行结果

【题目分析】本题主要考查多态、虚函数的知识。各种职员的类图如图 11-25 所示。主函数部分的程序流程图如图 11-26 所示。

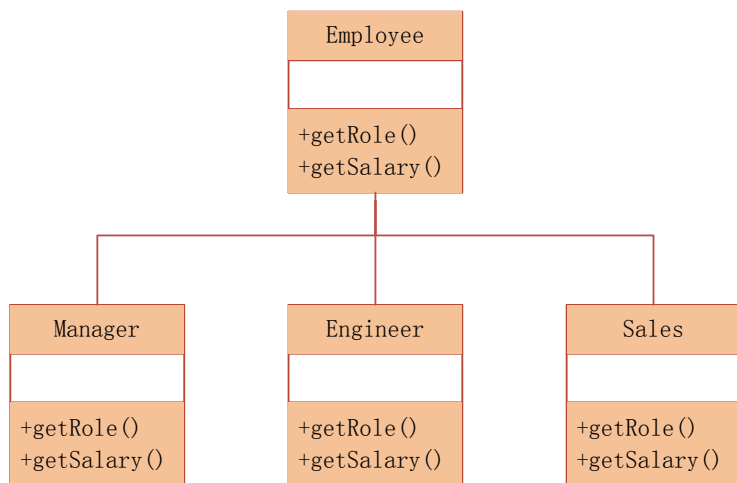


图 11-25 各种职员类图

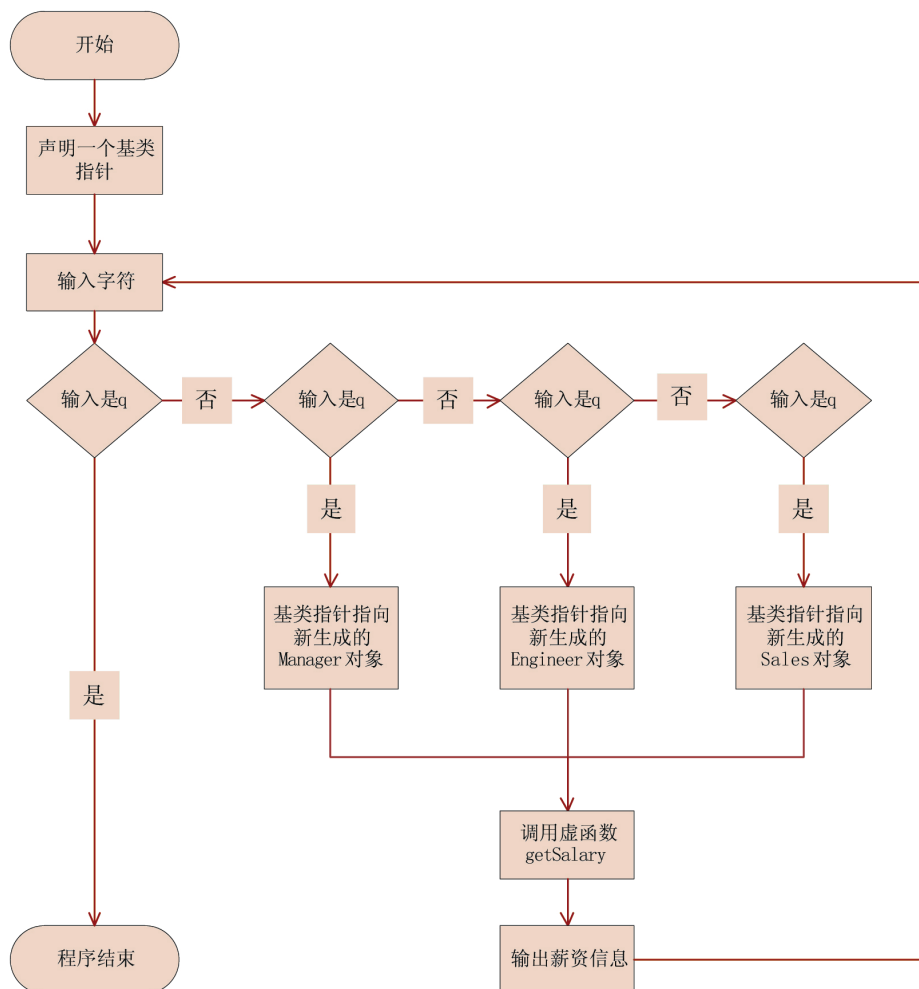


图 11-26 输出各种职员的薪水流程图

**【关键代码】**

```
class Employee
{
public:
    virtual double getSalary();
};
double Employee::getSalary()
{
    return 0.0;
}

class Manager:public Employee
{
    virtual double getSalary();
};
double Manager::getSalary()
{
    return 10000;
}

class Engineer:public Employee
{
    virtual double getSalary();
};
double Engineer::getSalary()
{
    return 8000;
}

class Sales:public Employee
{
    virtual double getSalary();
};
double Sales::getSalary()
{
    return 6000;
}
```

**主函数:**

```
cout<<"请输入[m:经理,e:工程师,s:销售,q:结束]: ";
Employee *pObj=NULL;
string role;
cin>>role;
while("q"!=role)
{
    if("m"==role)
    {
        pObj=new Manager();
    }
    else if("e"==role)
    {
        pObj=new Engineer();
    }
    else if("s"==role)
    {
        pObj=new Sales();
    }
    cout<<"薪资: "<<pObj->getSalary()<<endl<<endl;
    delete pObj;
    pObj=NULL;
}
```





```

        cout<<"[m,e,s,q]: ";
        cin>>role;
    }

```

【题目 11-2】定义一个抽象类，然后将这个类作为基类，再派生两个子类，最后调用其中的虚函数显示输出数据。

【题目分析】本题要求读者熟悉类继承、抽象类、虚函数的相关知识。

【关键代码】

```

class base
{
public:
    virtual void disp()
    {
        cout<<"hello base"<<endl;
    }
};

class child1:public base
{
public:
    void disp()
    {
        cout<<"hello child1"<<endl;
    }
};

class child2:public base
{
public:
    void disp()
    {
        cout<<"hello child2"<<endl;
    }
};

```

【题目 11-3】定义两个抽象类 Point，其中包括一个纯虚函数 getArea()，然后派生两个子类 CRect 和 CTrigon，最后在主函数中初始化并调用该虚函数输出结果。程序的运行结果如图 11-27 所示。

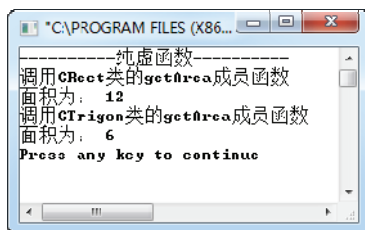


图 11-27 运行结果

【题目分析】本题要求读者掌握纯虚函数的相关知识。重点是掌握纯虚函数的作用及其实现方法。

【关键代码】

```

class CPoint
{
private:
    double m_w;
    double m_h;

```



```
public:
    CPoint(double w,double h);
    virtual double getArea()=0;
};
CPoint::CPoint(double w,double h)
{
    m_w=w;
    m_h=h;
}

class CRect:public CPoint
{
private:
    double m_w;
    double m_h;
public:
    CRect(double w,double h);
    double getArea();
};
CRect::CRect(double w,double h):CPoint(w,h)
{
    m_w=w;
    m_h=h;
}
double CRect::getArea()
{
    cout<<"调用 CRect 类的 getArea 成员函数"<<endl;
    return m_w*m_h;
}

class CTrigon:public CPoint
{
private:
    double m_f;
    double m_h;
public:
    CTrigon(double f,double h);
    double getArea();
};
CTrigon::CTrigon(double f,double h):CPoint(f,h)
{
    m_f=f;
    m_h=h;
}
double CTrigon::getArea()
{
    cout<<"调用 CTrigon 类的 getArea 成员函数"<<endl;
    return m_f*m_h/2;
}

cout<<"-----纯虚函数-----"<<endl;
CPoint *p=NULL;
p=new CRect(3,4);
cout<<"面积为: "<<p->getArea()<<endl;
p=new CTrigon(3,4);
cout<<"面积为: "<<p->getArea()<<endl;
```

# 第 12 章 运算符重载

第 11 章介绍了多态的实现技术，主要包括函数的重载和虚函数。本章介绍实现多态的另外一个方法——运算符重载及类型转换。



## 12.1 运算符重载概述

运算符重载实现的是编译时的多态，是对已有的运算符赋予多重含义，使同一个运算符作用于不同类型的数据时，导致不同类型的行为。

### 12.1.1 什么是运算符重载

C++预定义的运算符只是针对基本数据类型，而对于自定义的数据类型，比如类，却没有类似的运算符。于是，引入运算符重载的目的如图 12-1 所示。

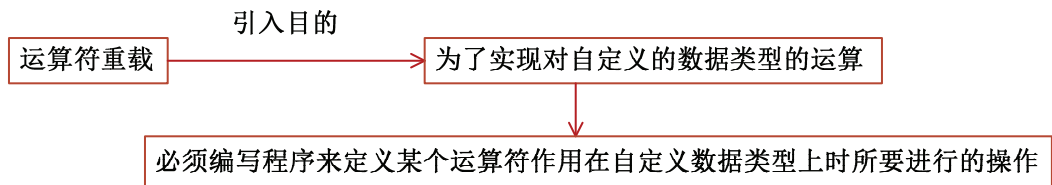


图 12-1 引入运算符重载的目的

运算符重载是运用函数重载的方法，对 C++提供的标准运算符重新定义，以完成某种特定的操作。

C++中的每个运算符对应着一个运算符函数，在实现过程中，把指定的运算表达式中的运算符转化为对运算符函数的调用，而表达式中的运算对象转化为运算符函数的实参，这个过程是在编译阶段完成的，如图 12-2 所示的例子。

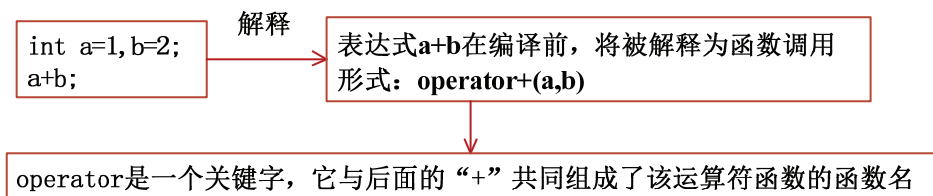


图 12-2 编译阶段转换



因此，可以将运算符重载看做是一种特殊的函数重载。运算符函数定义的一般形式如图 12-3 所示。

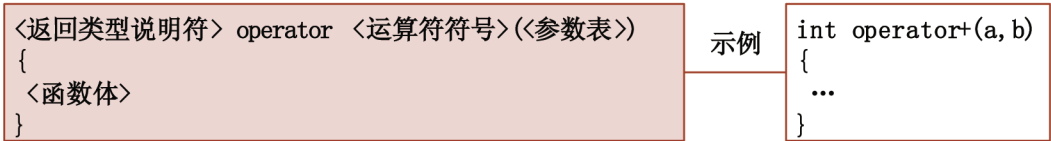


图 12-3 运算符函数定义的一般形式

12.1.2 运算符重载的特点

尽管运算符重载是一种特殊的函数重载，但相比函数重载，运算符重载有自身的一些特点。运算符重载使用 `operator` 关键字对重载函数进行标识和定义，其有 3 种形式，如表 12-1 所示。

表 12-1 3 种运算符的operator表示形式

运算符分类	常规表示	operator表示形式	参数个数
中缀	a+b	operator +(a,b)	二元
前缀	-a	operator -(a)	一元
后缀	a++	operator ++(a)	一元

C++运算符重载不允许用户自己定义新的运算符，只能对已有的运算符进行重载。在 C++ 中，绝大部分运算符允许重载，只有几个不能被重载，如图 12-4 所示。



图 12-4 几个不能被重载的运算符

此外，运算符重载时参数个数必须固定，即重载函数的参数个数与标准运算符保持一致，如图 12-5 所示。

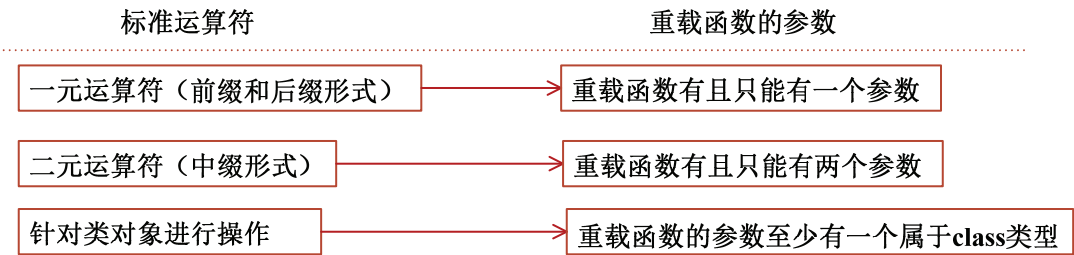


图 12-5 重载函数的参数个数与标准运算符要保持一致

【示例 12-1】下面通过一个示例来理解在实际的程序中是如何进行运算符重载的，其实现代码及结果如图 12-6 所示。



```

#include <iostream.h>
class Add
{
public:
    int operand;
    Add()
    {
        operand=0;
    }
    Add(int value)
    {
        operand=value;
    }
};
Add operator+(Add a, int b)
{
    Add sum;

    sum.operand=a.operand + b;
    return sum;
}
int main()
{
    Add a(5), b;

    b=a + 8;
    cout<<"the sum is: "<<b.operand<<endl;
    return 0;
}

```

定义类 Add

定义构造函数

重载构造函数

重载 “+” 运算符，操作 Add 类

创建对象

实现类中数据成员与整型变量相加

创建对象

调用重载后的 “+” 运算符

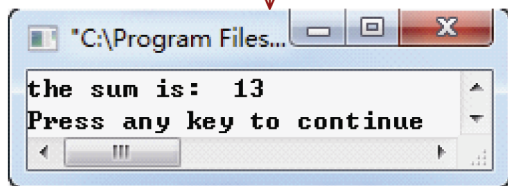


图 12-6 运算符重载实例

关于重载操作符的一些限制如下：

- (1) 不能改变重载运算符的优先级和结合性。
- (2) 默认重载运算符不能用做参数。
- (3) 不能改变原有运算符的参数个数。
- (4) 不能创建新的运算符。
- (5) 并不是所有的运算符都支持重载，但是大多数支持。



## 12.2 运算符重载形式

运算符的重载形式有两种：一种是重载为类的成员函数；另一种是重载为类的友元函数。对于每一种重载形式，由于运算符不同，都可以分为双目运算符和单目运算符的实现。



### 12.2.1 运算符重载为类的成员函数

将运算符重载为类的成员函数，称为运算符成员函数。实际使用时，总是通过该类的对象访问重载的运算符。运算符成员函数在类内进行声明，在类外进行定义，一般形式如图 12-7 所示。

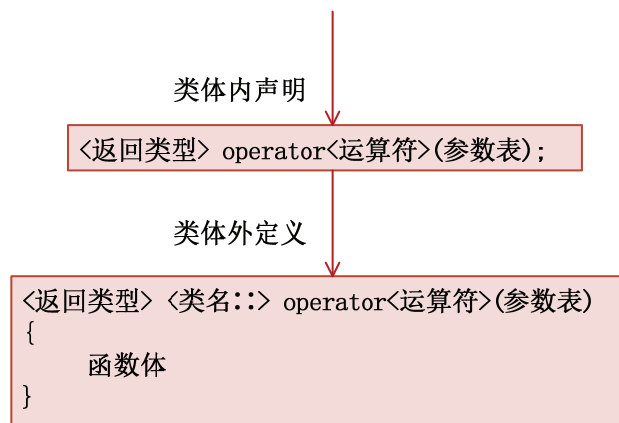


图 12-7 运算符成员函数在类内进行声明，在类外进行定义的一般形式

#### 1. 双目运算符重载为成员函数

双目运算符重载为成员函数时，左操作数是访问该重载运算符的对象本身的数据，此时成员运算符函数只有一个参数，如图 12-8 所示。

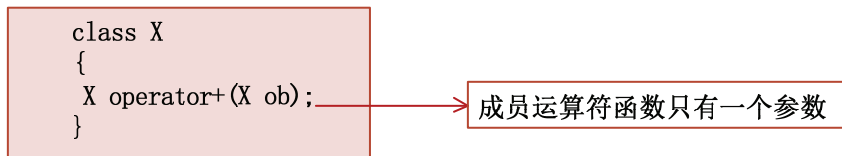


图 12-8 双目运算符重载为成员函数

双目运算符重载为成员函数后，就可以在主函数或其他类中进行调用。在 C++ 中，一般有显式和隐式两种调用方法，如图 12-9 所示。

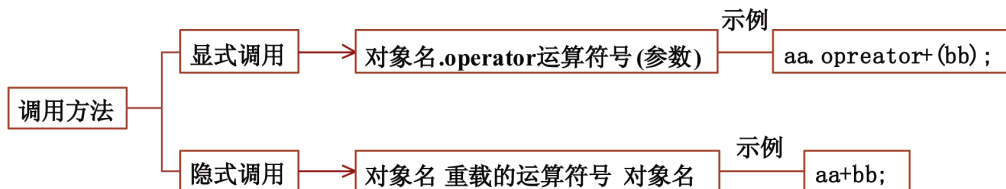


图 12-9 运算符成员函数的调用方法

【示例 12-2】下面的程序对双目运算符进行了重载，并在主函数中调用这些运算符进行具体操作，其分别采用了显式调用和隐式调用两种方式。实现代码及结果如图 12-10 所示。

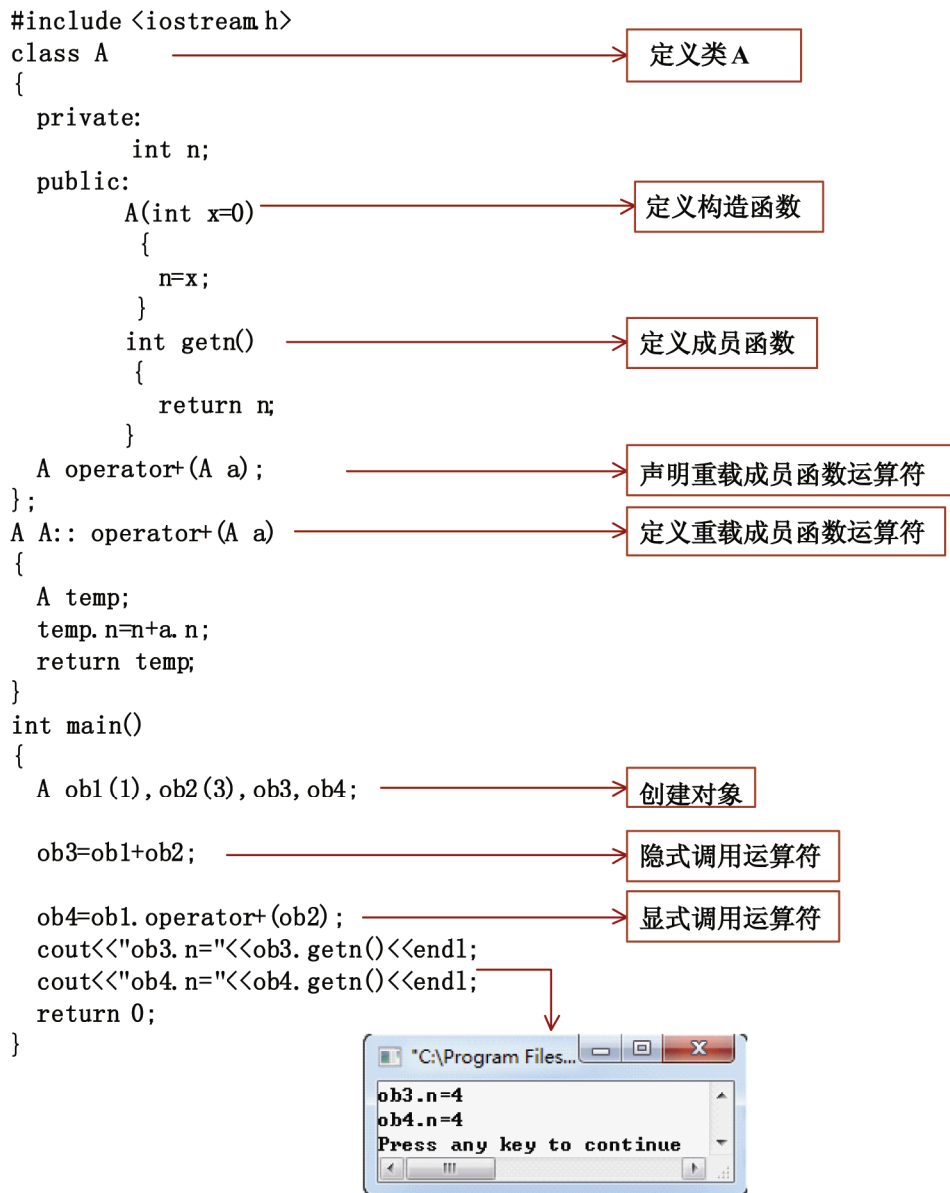


图 12-10 双目运算符重载为成员函数实例

## 2. 单目运算符重载为成员函数

单目运算符重载为成员函数时，操作数是访问该重载运算符对象本身的数据，由 this 指针指出，此时成员运算符函数没有参数，如图 12-11 所示。

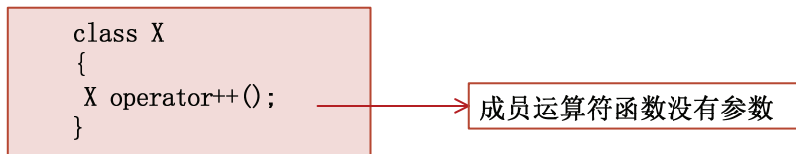


图 12-11 单目运算符重载为成员函数



与双目运算符的重载类似，单目运算符重载为成员函数后，在调用时也有显式和隐式两种调用方法，如图 12-12 所示。

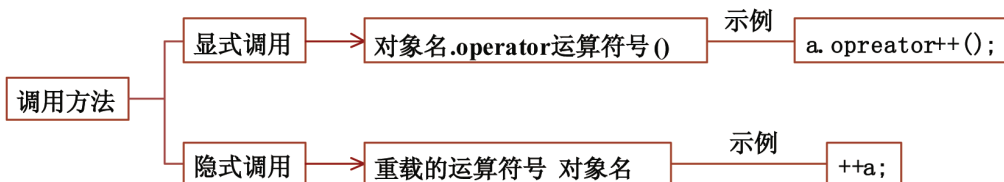


图 12-12 单目运算符重载为成员函数的调用方法

【示例 12-3】下面的程序对单目运算符进行重载，在调用该运算符时也通过显式调用和隐式调用两种方式来实现。其实现代码及结果如图 12-13 所示。

```
#include <iostream h>
class A
{
private:
    int n;
public:
    A(int x=0)
    {
        n=x;
    }
    int getn()
    {
        return n;
    }
    A operator++();
};
A A:: operator++()
{
    ++n;
    return *this;
}
int main()
{
    A ob(1);
    ++ob;
    cout<<"ob.n="<<ob.getn()<<endl;

    ob.operator++();
    cout<<"ob.n="<<ob.getn()<<endl;
    return 0;
}
```

定义类 A

定义构造函数

定义成员函数

声明重载成员函数运算符

定义重载成员函数运算符

创建对象

隐式调用运算符

显式调用运算符

图 12-13 单目运算符重载为成员函数实例





### 12.2.2 运算符重载为类的友元函数

将重载的运算符成员函数定义为类的友元函数，称为友元运算符函数。友元运算符函数不是类的成员，不属于任何一个类对象，所以没有 `this` 指针。因此，重载双目运算符时要有两个参数，重载单目运算符时要一个参数。友元运算符函数在类内进行声明，在类外进行定义，一般形式如图 12-14 所示。

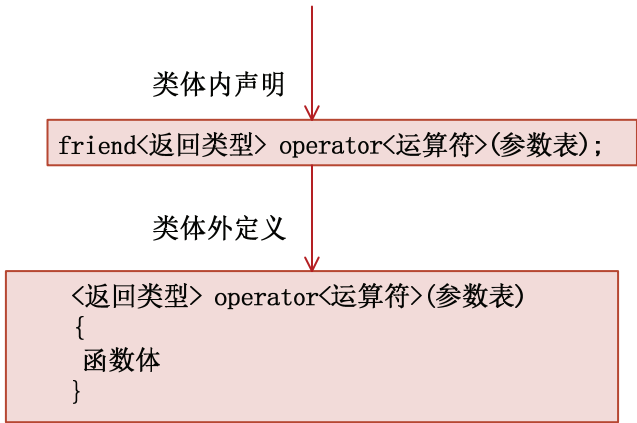


图 12-14 友元运算符函数在类内进行声明，在类外进行定义的一般形式

#### 1. 双目运算符重载为友元函数

双目运算符重载为友元函数时，由于没有 `this` 指针，所以两个操作数都要通过友元运算符函数的参数指出。与运算符重载为类的成员函数类似，双目运算符重载为友元函数后，其调用也有显式和隐式两种方法，如图 12-15 所示。

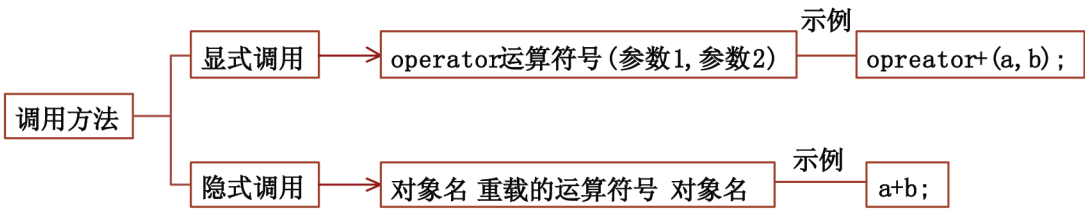


图 12-15 双目运算符重载为友元函数的调用方法

【示例 12-4】下面的程序对双目运算符进行了重载，并在主函数中调用这些运算符进行具体操作，分别采用了显式调用和隐式调用两种方式。其实现代码及结果如图 12-16 所示。



```
#include <iostream h>
class A
{
    private:
        int a, b;
    public:
        A(int x=0, int y=0)
        {
            a=x;
            b=y;
        }
        int geta()
        {
            return a;
        }
        int getb()
        {
            return b;
        }
}

friend A operator+(A p, A q);

A operator+(A p, A q)
{
    A temp;
    temp.a=p.a+q.a;
    temp.b=p.b+q.b;
    return temp;
}

int main()
{
    A ob1(1, 2), ob2(3, 4), ob3, ob4;

    ob3=ob1+ob2;

    ob4=operator+(ob1, ob2);

    cout<<"ob3. a="<<ob3.geta()<<"    ob3. b="<<ob3.getb()<<endl;
    cout<<"ob4. a="<<ob4.geta()<<"    ob4. b="<<ob4.getb()<<endl;
    return 0;
}
```

定义类 A

定义构造函数

定义成员函数

声明运算符重载为友元函数

定义友元函数

创建对象

隐式调用运算符

显式调用运算符

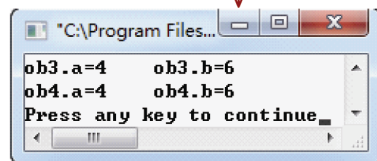


图 12-16 双目运算符重载为友元函数实例

双目运算符重载为友元函数和双目运算符重载为成员函数的根本区别在于其操作数的个数不同，前者需要指定两个参数，而后者定义时只需一个参数。

## 2. 单目运算符重载为友元函数

与单目运算符重载为成员函数不同，单目运算符重载为友元函数时，由于没有 `this` 指针，所以操作数要通过友元运算符函数的参数指出。单目运算符重载为友元函数后也有显式和隐式



两种调用方法，如图 12-17 所示。

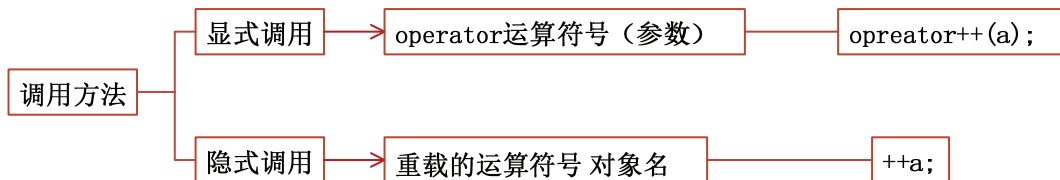


图 12-17 单目运算符重载为友元函数的调用方法

【示例 12-5】下面的程序对单目运算符进行重载，在调用该运算符时也通过显式调用和隐式调用两种方式来实现，其实现代码及结果如图 12-18 所示。

```

#include <iostream.h>
class A                                定义类 A
{
private:
    int n;
public:
    A(int x=0)                          定义构造函数
    {
        n=x;
    }
    int getn()                          定义成员函数
    {
        return n;
    }
};
friend A operator++(A &a);              声明单目运算符重载为友元函数
A operator++(A &a)                      定义重载友元函数运算符
{
    ++ a.n;
    return a;
}
int main()
{
    A ob(3);                            创建对象
    cout<<"ob.n="<<ob.getn()<<endl;

    ++ob;                               隐式调用运算符
    cout<<"ob.n="<<ob.getn()<<endl;

    operator++(ob);                     显式调用运算符
    cout<<"ob.n="<<ob.getn()<<endl;
    return 0;
}
  
```

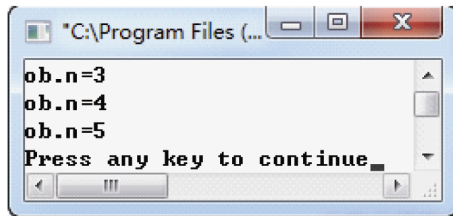


图 12-18 单目运算符重载为友元函数实例



在将运算符重载为友元函数时，有几个运算符不能用友元函数重载，如图 12-19 所示。

不能用友元函数重载的运算符

=、()、[]、->

图 12-19 不能用友元函数重载的运算符



**注意：**使用友元函数重载单目运算符“++”和“--”时，由于要改变操作数自身的值，所以应采用引用参数传递操作数，否则，会出现错误。

## 12.2.3 运算符成员函数与友元运算符函数的比较

前面介绍了将运算符重载为成员函数和友元函数，图 12-20 对这两种方法进行了比较。

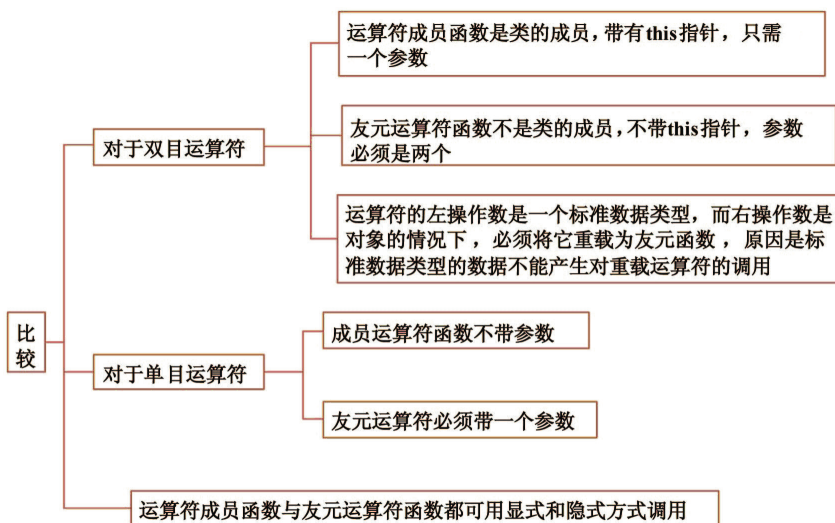


图 12-20 运算符重载为成员函数和友元函数方法的比较

总的来说，将运算符重载为成员函数还是重载为友元函数，要根据实际情况和使用习惯决定。一般而言，对于双目运算符重载为友元函数较好，若运算符的操作数特别是左操作数需要进行类型转换，必须重载为友元运算符函数。若一个运算符需要修改对象的状态，则选择运算符成员函数较好。



## 12.3 特殊运算符重载

特殊运算符是指除如上提到的一些常用运算符外对于 C++ 所支持的特殊运算符的重载。本节主要介绍“++”和“--”、“=”、“[]”4 种特殊运算符的重载。

### 12.3.1 “++”和“--”重载

运算符“++”和“--”的重载要区分前置和后置两种形式。例如，表达式“a++”和表达式“++a”是不一样的。

如果不区分前置和后置，则使用 `operator++()` 或 `operator--()` 即可，如图 12-21 所示。

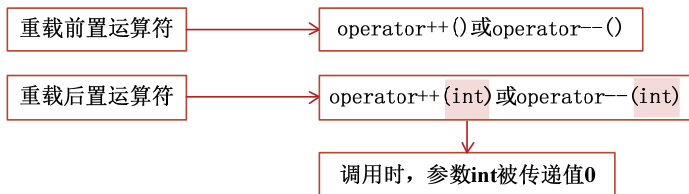


图 12-21 区分前置和后置

前增运算符和前减运算符重载为类的成员函数时不能带参数。

后增运算符和后减运算符重载为类的成员函数时必须带一个参数。

【示例 12-6】下面的程序对“++”和“--”进行重载，其实现代码及结果如图 12-22 所示。

```

#include <iostream.h>
class A
{
private:
    int n;
public:
    A(int x=0)
    {
        n=x;
    }
    int getn()
    {
        return n;
    }
    A operator++();
    A operator--(int);
};

A A:: operator++()
{
    ++n;
    return *this;
}

A A:: operator--(int)
{
    n--;
    return *this;
}

int main()
{
    A ob(3);
    cout<<"ob.n="<<ob.getn()<<endl;

    ++ob;
    cout<<"ob.n="<<ob.getn()<<endl;

    ob.operator--(0);
    cout<<"ob.n="<<ob.getn()<<endl;
    return 0;
}
  
```

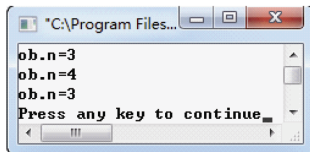


图 12-22 “++”和“--”重载运算符实例



**注意：**声明和定义后置的“++”或“--”等运算符重载，必须含有形式参数，在调用时一般为指定实参 0。



## 12.3.2 赋值运算符“=”重载

在进行赋值运算符“=”重载的应用前，先了解一下赋值运算符“=”在实际程序中的运行情况。事实上，对于任何一个类，如果没有用户自定义的赋值运算符函数，系统会自动为其生成一个默认的赋值运算符函数，以完成数据成员之间的复制，如图 12-23 所示的程序段。

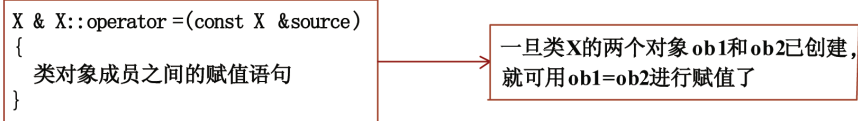
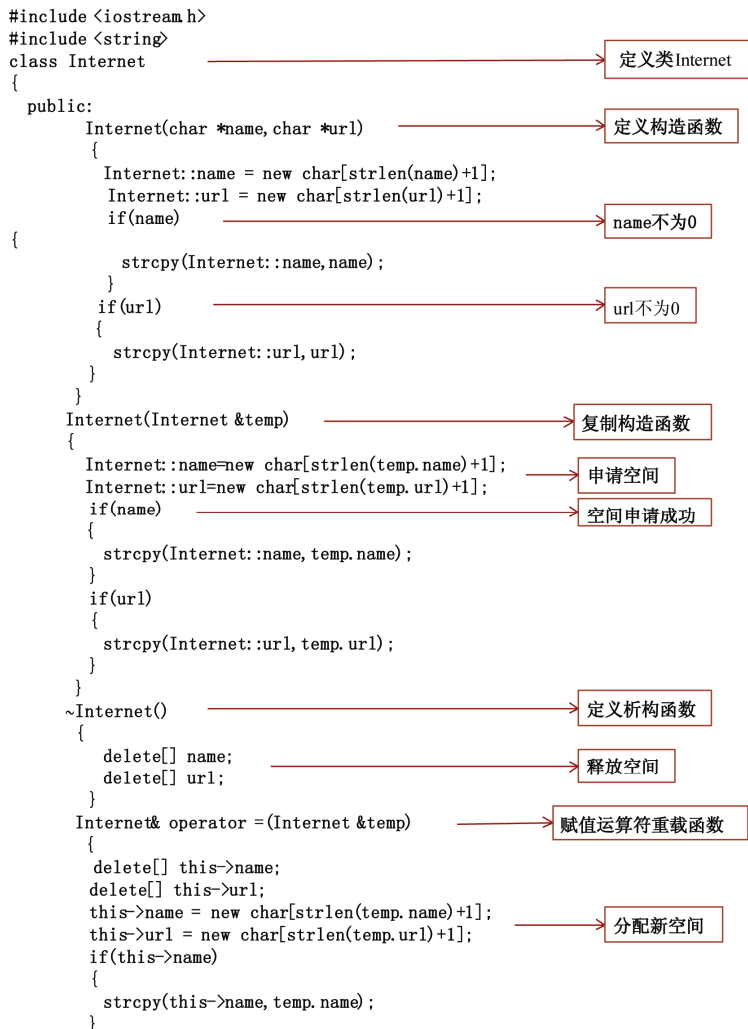


图 12-23 默认的赋值运算符函数

通常，默认的赋值运算符函数就可完成赋值任务，但在某些特殊情况下，例如，类中有一种指针类的形式，如果使用默认的赋值运算符函数就会产生错误。此时，就必须显式地定义一个赋值运算符重载函数，使参与赋值的两个对象有各自的存储空间，以解决这个问题。

【示例 12-7】下面的程序实现赋值运算符“=”的重载，重载后的赋值运算符能够实现网站名称和地址字符串之间的赋值。其实现代码及结果如图 12-24 所示。





```

if(this->url)
{
    strcpy(this->url, temp.url);
}
return *this;
}
public:
    char *name;
    char *url;
};
int main()
{
    Internet a("pxc", "www.pxc.jx.cn");
    Internet b = a;
    cout<<b.name<<endl<<b.url<<endl;
    Internet c("Tsinghua", "www.tsinghua.edu.cn");
    b = c;
    cout<<b.name<<endl<<b.url<<endl;
    return 0;
}

```

创建对象

b对象还不存在，调用复制构造函数

b对象已经存在，调用赋值运算符重载函数

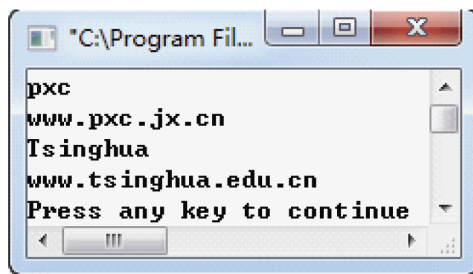


图 12-24 赋值运算符“=”重载实例



**注意：**赋值运算符只能重载为运算符成员函数，不能重载为友元运算符函数，而且赋值运算符重载后不能被继承。

### 12.3.3 下标运算符“[]”重载

下标运算符 `operator[]` 通常用来访问数组中的某个元素，可以看做是一个双目运算符，第一个运算符是数组名，第二个运算符是数组下标。在类对象中，可以重载下标运算符，用它来定义相应对象的下标运算。下标运算符定义的一般形式如图 12-25 所示。

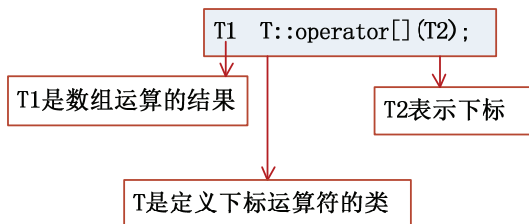


图 12-25 下标运算符定义的一般形式

在实际的程序中，可以通过下面两种方式来调用下标运算符，如图 12-26 所示。

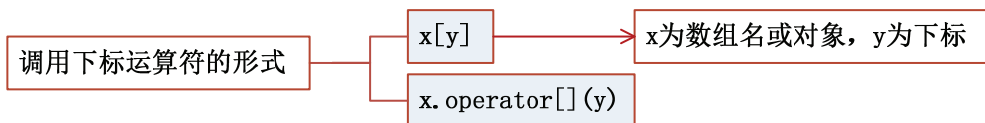


图 12-26 调用下标运算符的形式

【示例 12-8】下面的程序重载了下标运算符，其实现代码及结果如图 12-27 所示。

```
#include <iostream.h>
class IntArray —————→ 定义类IntArray
{
    int *a;
    int sz;
public:
    IntArray(int size) —————→ 定义构造函数
    {
        sz=size;
        a=new int[size];
    }
    int &operator[] (int i) —————→ 重载下标运算符 []
    {
        if (i<0 || i>=sz) —————→ 判断是否越界
        {
            cout<<"error"<<endl;
        }
        return a[i];
    }
};
int main()
{
    IntArray a(5); —————→ 创建对象

    a[3]=0; —————→ 赋值，不越界
    cout<<"a[3]= " <<a[3]<<endl;

    a.operator[] (3)=0; —————→ 赋值，不越界
    cout<<"a.operator[] (3)= " <<a[3]<<endl;

    cout<<"a[6]= "; —————→ 赋值，越界
    a.operator[] (6)=6;
    return 0;
}
```

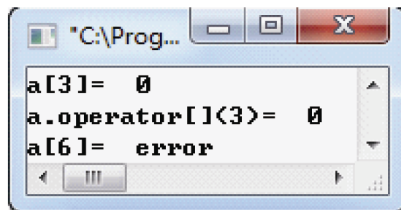


图 12-27 下标运算符“[]”重载实例



**注意：**C++不允许把下标运算符函数作为外部函数来定义，只能是非静态的成员函数。





## 12.4 类类型转换

类类型是指某个对象的数据类型为类，而不是标准的数据类型。在 C++ 中，标准的数据类型与类类型之间的转换有 3 种方法，如图 12-28 所示。

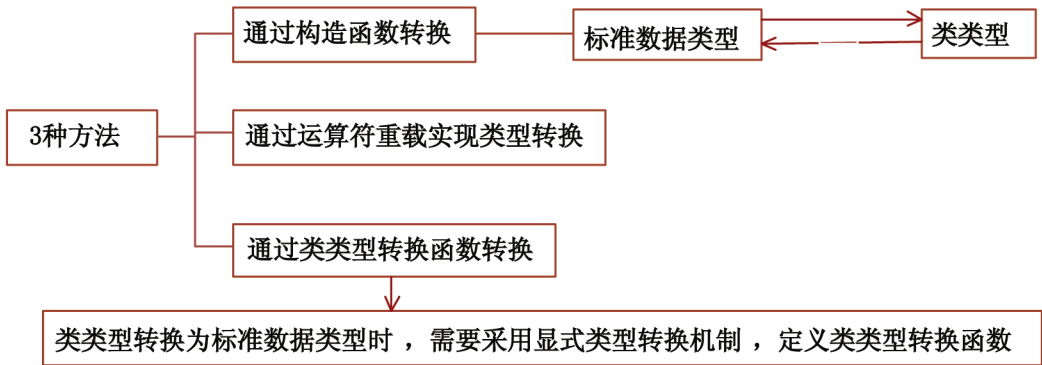


图 12-28 标准的数据类型与类类型之间的转换的 3 种方法

C++ 中，定义一个类的类型转换函数的一般形式如图 12-29 所示。

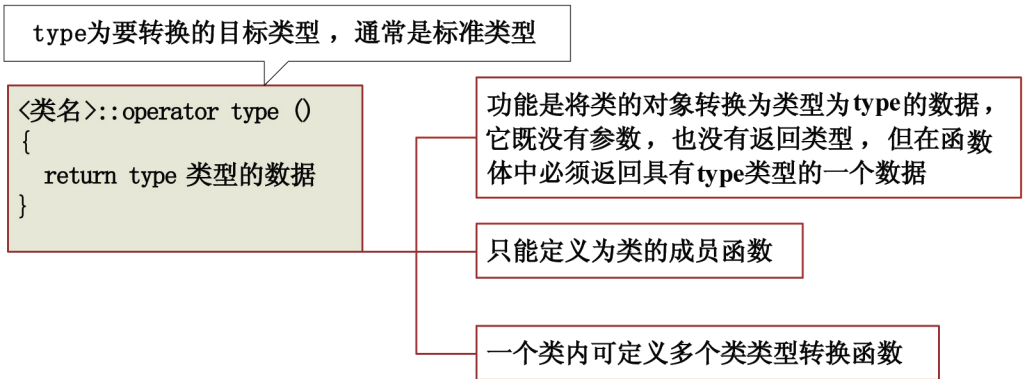


图 12-29 类的类型转换函数



**提示：**在实际的程序中，使用最多的还是通过运算符重载来实现类型的转换。

【示例 12-9】下面通过示例来进一步了解通过运算符重载来实现类型的转换，其实现代码及结果如图 12-30 所示。



```

#include <iostream h>
class Test
{
public:
    Test(int a = 0)
    {
        cout<<this<<": "<<"载入构造函数！"<<a<<endl;
        Test::a = a;
    }
    Test(Test &temp)
    {
        cout<<"载入复制构造函数！"<<endl;
        Test::a = temp.a;
    }
    ~Test()
    {
        cout<<this<<": "<<"载入析构函数！"<<this->a<<endl;
        cin.get();
    }
    operator int()
    {
        cout<<this<<": "<<"载入转换运算符函数！"<<this->a<<endl;
        return Test::a;
    }
public:
    int a;
};

int main()
{
    Test b(99);
    cout<<"b的内存地址："<<&b<<endl;
    cout<<(int)b<<endl;
    return 0;
}

```

定义类

定义构造函数

定义复制构造函数

定义析构函数

转换运算符重载

创建对象

强转换

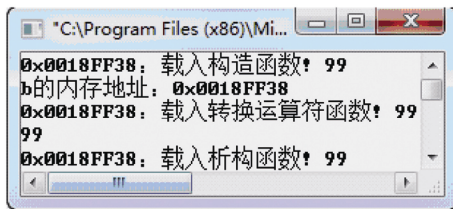


图 12-30 运算符重载来实现类型的转换实例



## 12.5 小结

本章主要讲述了实现多态的一种形式——运算符重载。重点讲解了运算符重载的两种形式及几个特殊运算符重载。最后简单讲述了类类型转换



## 12.6 习题

【题目 12-1】自定义一个四则运算类，其对象可以对两个数进行四则运算，最后输出结果。程序的运行结果如图 12-31 所示。

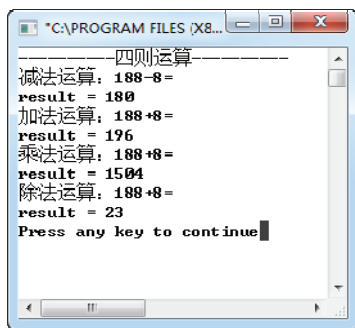


图 12-31 运行结果

【题目分析】本题主要考查运算符重载的相关知识。

【关键代码】

```
class CCalc
{
private:
    int n;
public:
    CCalc(int x)
    {
        n=x;
    }
    CCalc operator + (const CCalc &);
    CCalc operator - (const CCalc &);
    CCalc operator * (const CCalc &);
    CCalc operator / (const CCalc &);
    void disp()
    {
        cout<<"result = "<<n<<endl;
    }
};

CCalc CCalc::operator +(const CCalc &CC)
{
    return CCalc(n+CC.n);
}
CCalc CCalc::operator -(const CCalc &CC)
{
    return CCalc(n-CC.n);
}
CCalc CCalc::operator *(const CCalc &CC)
{
    return CCalc(n*CC.n);
}
CCalc CCalc::operator /(const CCalc &CC)
{
    if(CC.n!=0)
    {
        return CCalc(n/CC.n);
    }
    cout<<"不能进行 0 值除法."<<endl;
    return CCalc(0);
}

cout<<"-----四则运算-----"<<endl;
CCalc cc1(188);
CCalc cc2(8);
CCalc cc3(0);
```



```
cout<<"减法运算: 188-8= "<<endl;
cc3=cc1-cc2;
cc3.display();
cout<<"加法运算: 188+8= "<<endl;
cc3=cc1+cc2;
cc3.display();
cout<<"乘法运算: 188*8= "<<endl;
cc3=cc1*cc2;
cc3.display();
cout<<"除法运算: 188/8= "<<endl;
cc3=cc1/cc2;
cc3.display();
```

【题目 12-2】在数学中有复数的概念，复数是由实数部分和虚数部分组成的。在 C++ 中并没有复数数据类型。请读者定义一个表示复数的类，类中用两个浮点数来分别表示实数部分和虚数部分，并为这个复数类以成员函数形式重载运算符加、减、乘、除、取反、前置++和后置++。程序的运行结果如图 12-32 所示。

```
"C:\Program Files (x86)\C++\gfaDe..
请依次输入两个复数的实部和虚部:
复数c1:
1.0
2.0
复数c2:
3.0
4.0
复数的减法运算: c1-c2 = -2 + i*-2
复数的取反运算: -c1 = -1 + i*-2
复数的加法运算: c1+c2 = 4 + i*6
复数的乘法运算: c1*c2 = -5 + i*10
复数的除法运算: c1/c2 = 0.36 + i*0.16
复数c3:
1.0
1.0
复数c4:
5.0
5.0
复数的前置++运算: ++c3 = 2 + i*2
复数的后置++运算: c4++ = 5 + i*5
Press any key to continue
```

图 12-32 运行结果

【题目分析】本题考查运算符重载的知识。

【关键代码】

```
class complex
{
private:
    double real,imag;
public:
    complex(double r=0.0,double i=0.0)
    {
        real=r;
        imag=i;
    }
    complex operator + (const complex &);
    complex operator - (const complex &);
    complex operator - ();
    complex operator * (const complex &);
    complex operator / (const complex &);
    complex& operator ++ ();
    complex operator ++(int);
    void disp()
```



```

    {
        cout<<real<<" + "<<"i*"<<imag<<endl;
    }
};

complex complex::operator +(const complex& CC)
{
    return complex(real+CC.real,imag+CC.imag);
}
complex complex::operator -(const complex& CC)
{
    return complex(real-CC.real,imag-CC.imag);
}
complex complex::operator -()
{
    return complex(-real,-imag);
}
complex complex::operator *(const complex& CC)
{
    return complex(real*CC.real-imag*CC.imag,real*CC.imag+imag*CC.real);
}
complex complex::operator /(const complex& CC)
{
    return
complex((real*CC.real+imag*CC.imag)/(CC.real*CC.real+CC.imag*CC.imag),(imag*CC.real
-real*CC.imag)/(CC.real*CC.real+CC.imag*CC.imag));
}
complex& complex::operator ++()
{
    real+=1;
    imag+=1;
    return (*this);
}
complex complex::operator ++(int)
{
    complex ctemp=*this;
    ++(*this);
    return ctemp;
}

```

### 主函数:

```
cout<<"请依次输入两个复数的实部和虚部: "<<endl;
```

```

complex cxRes;
double r,i;
cout<<"复数 cx1: "<<endl;
cin>>r;
cin>>i;
complex cx1(r,i);
cout<<"复数 cx2: "<<endl;
cin>>r;
cin>>i;
complex cx2(r,i);

cout<<"复数的减法运算: cx1-cx2 = ";
cxRes=cx1-cx2;
cxRes.disp();

cout<<"复数的取反运算:  -cx1 = ";
cxRes=-cx1;
cxRes.disp();

cout<<"复数的加法运算: cx1+cx2 = ";

```



```
cxRes=cx1+cx2;
cxRes.disp();

cout<<"复数的乘法运算: cx1*cx2 = ";
cxRes=cx1*cx2;
cxRes.disp();

cout<<"复数的除法运算: cx1/cx2 = ";
cxRes=cx1/cx2;
cxRes.disp();

cout<<"复数 cx3: "<<endl;
cin>>r;
cin>>i;
complex cx3(r,i);
cout<<"复数 cx4: "<<endl;
cin>>r;
cin>>i;
complex cx4(r,i);

cout<<"复数的前置++运算: ++cx3 = ";
cxRes=++cx3;
cxRes.disp();

cout<<"复数的后置++运算: cx4++ = ";
cxRes=cx4++;
cxRes.disp();
```

【题目 12-3】在【题目 12-2】上，以友元函数形式重载运算符。

【题目分析】考查运算符的重载形式：以友元函数形式重载运算符。

【关键代码】

```
class complex
{
private:
    double real,imag;
public:
    complex(double r=0.0,double i=0.0)
    {
        real=r;
        imag=i;
    }
    friend complex operator + (const complex &,const complex &);
    friend complex operator - (const complex &,const complex &);
    friend complex operator * (const complex &,const complex &);
    friend complex operator / (const complex &,const complex &);
    friend complex& operator ++ (complex &);
    friend complex operator ++(complex &,int);
    void disp()
    {
        cout<<real<<" + "<<"i*"<<imag<<endl;
    }
};

complex operator +(const complex & C1,const complex & C2)
{
    return complex(C1.real+C2.real,C1.imag+C2.imag);
}
complex operator -(const complex & C1,const complex & C2)
{
    return complex(C1.real-C2.real,C1.imag-C2.imag);
}
```



```
}
complex operator -(const complex & C1)
{
    return complex(-C1.real,-C1.imag);
}
complex operator *(const complex & C1,const complex & C2)
{
    return complex(C1.real*C2.real-C1.imag*C2.imag,
        C1.real*C2.imag+C1.imag*C2.real);
}
complex operator /(const complex & C1,const complex & C2)
{
    return
complex( (C1.real*C2.real+C1.imag*C2.imag) / (C2.real*C2.real+C2.imag*C2.imag),
        (C1.imag*C2.real-C1.real*C2.imag) / (C2.real*C2.real+C2.imag*C2.imag));
}
complex& operator ++(complex & C1)
{
    C1.real+=1;
    C1.imag+=1;
    return C1;
}
complex operator ++(complex & C1,int)
{
    complex ctemp=C1;
    ++C1;
    return ctemp;
}
```

# 第 13 章 类模板

类模板是一种对类型进行参数化和实现代码复用的技术。使用类模板可以提高数据的处理效率。本章将详细讲解类模板的有关知识。主要包括类模板的定义和使用、静态成员、友元、特化等内容。由于该内容较抽象，请读者仔细阅读。

## 13.1 什么是类模板

类模板就是类的模板，是对数据的类型进行了参数化处理。类模板实例化的含义如图 13-1 所示。

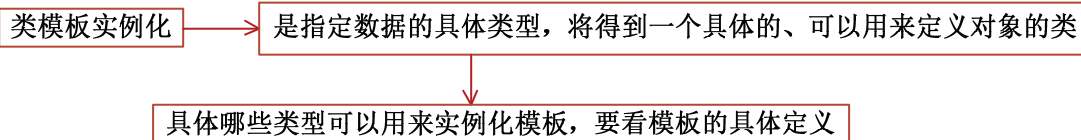


图 13-1 类模板实例化的含义



**说明：**类模板所能接受的具体类型参数都是有限制。

在程序中采用类模板的目的如图 13-2 所示。

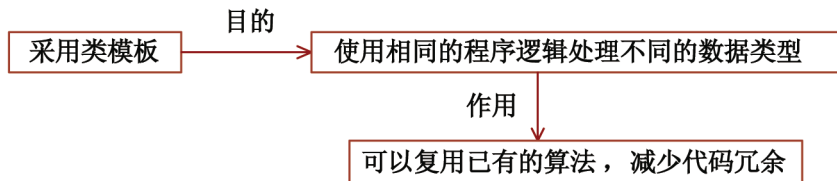


图 13-2 程序中采用类模板的目的

例如，设计一个栈的类。如果不采用类模板，则在设计之初就要确定栈中数据的类型。如果采用类模板，则只需通过类模板实例化实现。其两种方法如图 13-3 所示。





## 不采用类模板

```
class Stack
```

```
{
```

```
public:
```

```
void push( int i );
```

压入元素

```
int pop();
```

弹出元素

```
private:
```

```
enum { size = 100 };
```

栈的初始容量

```
int s_array[size];
```

保存元素的数组

```
int s_count;
```

元素个数

```
};
```

只能用于处理 **int** 型数据，或其他与 **int** 型数据存在转换关系的数据

## 采用类模板

```
template< typename T >
```

```
class Stack
```

```
{
```

```
public:
```

```
void push(T i);
```

```
T pop();
```

```
private:
```

```
enum { size = 100 };
```

```
T s_array[size];
```

```
int s_count;
```

```
};
```

实例化 Stack 模板

```
Stack<int> s1;
```

```
Stack<double> s2;
```

```
Stack<Shape> s3;
```

图 13-3 不采用类模板与采用类模板

由图 13-3 可以得出，使用类模板的主要优点如图 13-4 所示。

使用类模板的主要优点

一个类模板可以处理不同类型的数据

提高程序开发的效率，不需要为不同的类型写相同的逻辑

图 13-4 使用类模板的主要优点



**提示：**类模板是 C++ 的一个非常重要的特征，在 C++ 的标准模板库（Standard Template Library, STL）中提供了大量的类模板，有关 STL 的内容将在后面章节中详细讲述。



## 13.2 定义类模板

定义类模板其实和定义函数的差别不大，其核心都是定义一个算法逻辑。唯一不同的是类模板将其使用的类型进行了参数化，对不同类型的数据用同一算法逻辑。

### 13.2.1 语法

一个类模板的定义是关键字 `template` 作为开始的，其定义的一般语法形式如图 13-5 所示。

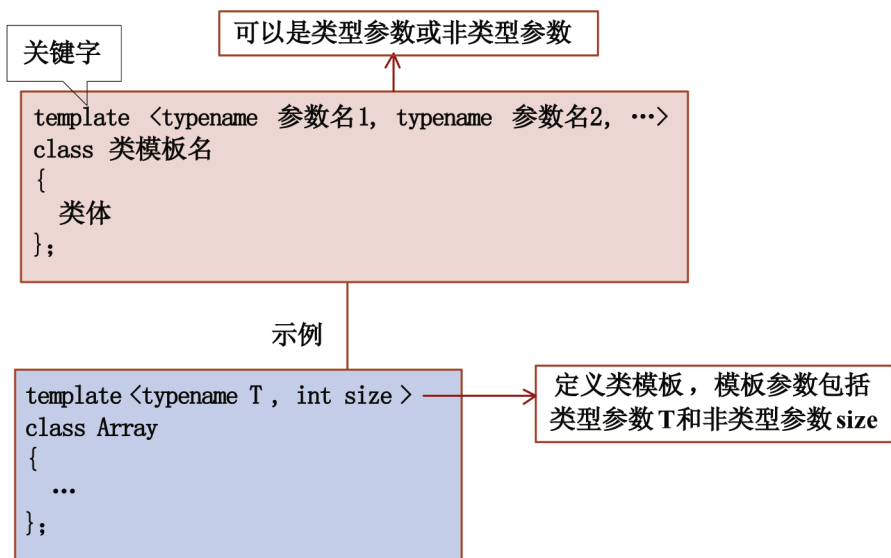


图 13-5 类模板定义的一般形式



**注意：**类模板定义最后的分号不可省，因为与类的定义一样，类模板的定义也是一条语句，需要一个分号作为语句的结束。

定义类模板的成员函数时，有两种方法，如图 13-6 所示。

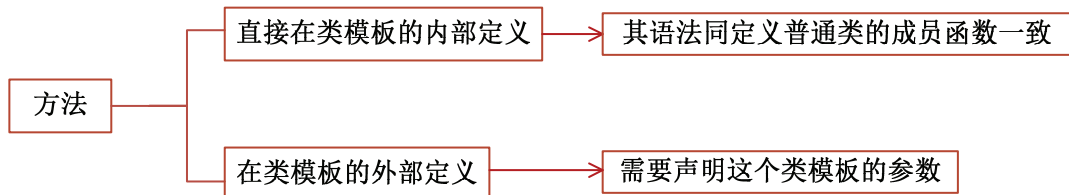


图 13-6 定义类模板的成员函数时的两种方法

在类的外部定义类模板的成员函数时的一般语法形式如图 13-7 所示。



```
template< 模板参数列表 >
返回类型 类模板名< 模板参数列表 >::函数名( 函数参数列表 )
{
    函数体
}
```

可以省略类型关键字

示例

```
template<typename T>
T stack<T>::pop()
{
    return s_array[ --s_count ]
}
```

图 13-7 在类的外部定义类模板的成员函数的一般形式

在定义类模板的数据成员和函数成员时，可以使用模板参数，具体情况如图 13-8 所示。

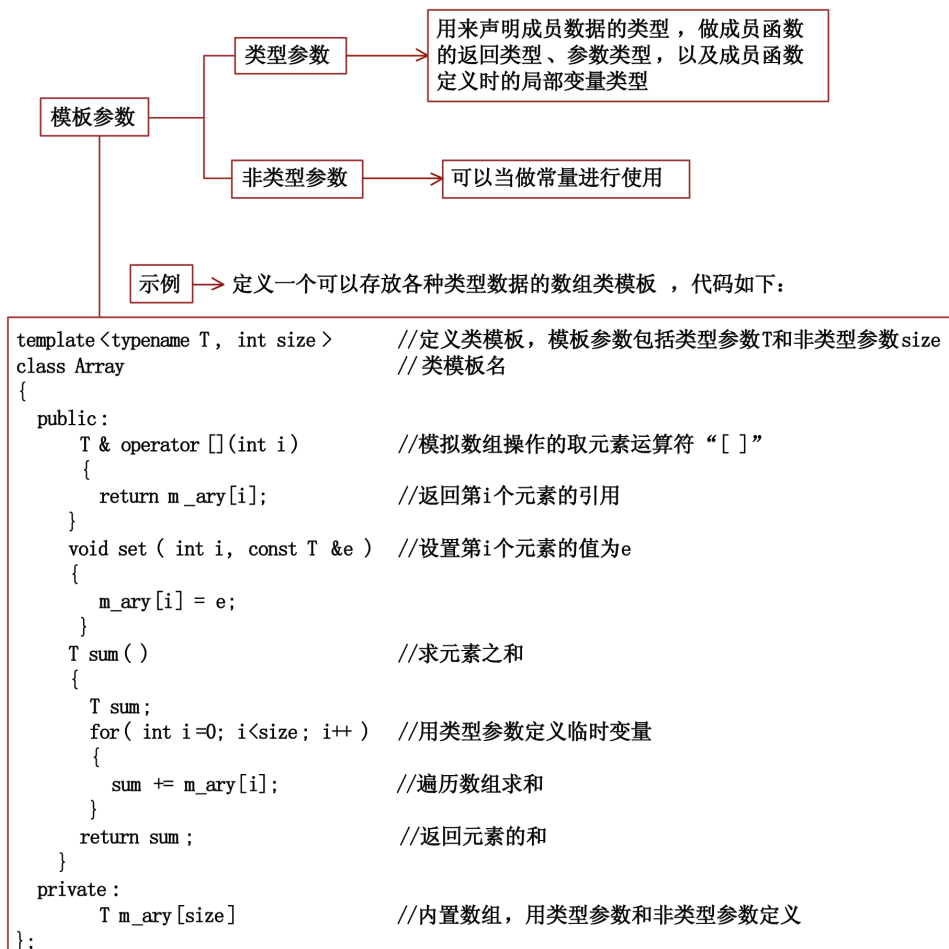


图 13-8 使用模板参数实例



### 13.2.2 非类型参数

类模板可以使用非类型参数。非类型的参数通常在模板定义过程中当做常量使用。例如，图 13-8 中的类模板 `Array` 的第二个模板参数 `size` 即是一个非类型的参数，并且该参数用来设定数组的长度。

声明一个类模板的非类型参数的一般语法形式如图 13-9 所示。



图 13-9 声明一个类模板的非类型参数的一般形式

对于带有非类型参数的类模板，实例化时需要用一个常量来指定这个参数，如图 13-10 所示的例子。

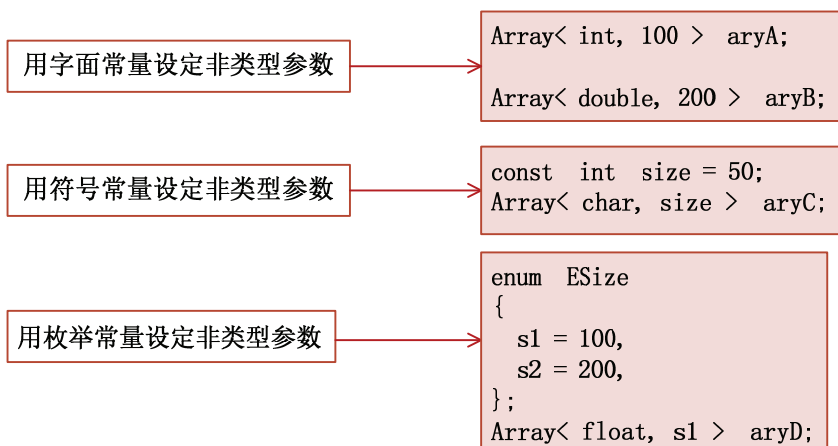


图 13-10 非类型参数实例化

### 13.2.3 模板参数的默认实参

定义类模板时，可以设定其参数的默认值，即参数的默认实参，包括默认类型参数和默认非类型参数。例如，对于 `Array` 模板，设定其默认保存的数据类型为 `int` 型，并且其数组尺寸的默认值为 100，如图 13-11 所示。

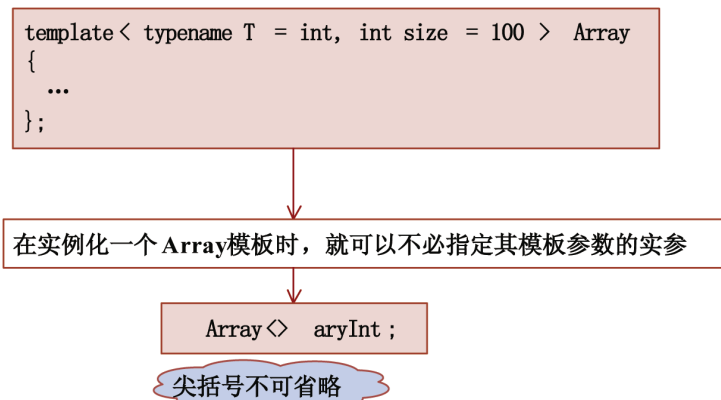


图 13-11 模板参数的默认实参



## 13.3 生成类模板的实例

类模板只是一个模板，不是实际的类。使用类模板时必须先实例化，即给类模板参数赋值，包括类型参数和非类型参数。本节将详细讨论如何实例化类模板。

### 13.3.1 类型参数的模板实例化

实例化类模板时，对于类型参数，可以用 C++ 内建的类型实例化，也可以用自定义的类型。类模板在实例化时没有参数推导机制，所有的模板参数必须指定，除非模板参数带有默认实参。例如，实例化类模板 `Array`，代码如图 13-12 所示。

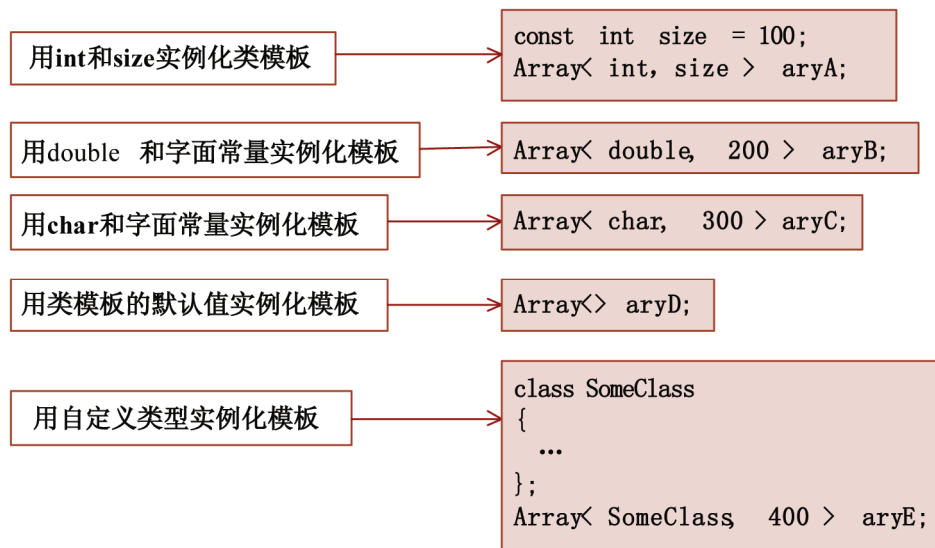


图 13-12 实例化类模板 `Array`

实例化类模板时，编译器用开发者指定的模板实参代替模板定义中的参数，创建出一种新的“类”实例，即一个类，然后可以用这个新的类去定义类对象。



**提示：**如果用自定义的类型来实例化一个类模板，则对自定义类型有一定的要求。

类模板实例可以当做普通的类使用，其定义的对象与一般类定义的对象在使用上没有什么差别。但要注意的是，类模板只在需要的时候才实例化，如图 13-13 所示。



程序中只是使用类模板的某个指针（或者引用），类模板不会实例化

示例

```
void Print(Stack<int>& vi)
{
    Stack<int>* pvi = &vi;
}
```

Stack<int>并不会被实例化

但通过这个指针（或引用）访问对象的数据成员或成员函数，则实例化

示例

```
void Print(Stack<int>& vi)
{
    Stack<int>* pvi = &vi;
    cout << pvi->Pop() << endl;
}
```

模板会被实例化

图 13-13 类模板只在需要的时候才实例化

### 13.3.2 非类型参数的模板实例化

对于模板的非类型参数，在实例化时只要指定其实参即可。但要注意的是，这个实参必须是一个常量，如图 13-14 所示的代码。

```
template<int* ptrInt> CRect {};
template<int size> CRectangle {};
const int c_size = 10;
int size = 10;
```

CRect<new int[10]> rect;

错误，new int[10]只有在运行时才能被计算

CRectangle<10> rectangleA

正确，10是常量

CRectangle<c\_size> rectangleB

正确，c\_size是常量表达式

CRectangle<size> rectangleC

错误，size的值在编译期不能被计算

CRect<&size> rectA

正确，地址是在编译期确定的

图 13-14 非类型参数的模板实例化

### 13.3.3 类模板示例

在实际开发中，类模板经常用来做一个可以处理各种类型数据的容器。例如，一个先入后出的栈，如果将栈定义成一个类模板，则可以用来处理各种数据，方便使用。

【示例 13-1】下面的程序定义了一个 Stack 类模板，其模板参数有类型参数和非类型参数，并用这个类模板来处理不同类型的数据，实现代码及结果如图 13-15 所示。



```

#include <iostream.h>
template<class T, int size = 100>
class Stack
{
public:
    Stack()
        : s_iCount(0)
        { }
    bool push(const T &i);
    bool pop(T &i);
    int get_size () const
    {
        return s_iCount;
    }
private:
    T s_iArray[size];
    int s_iCount;
};

template<class T, int size>
bool Stack<T, size>::push(const T &i)
{
    if (s_iCount < size)
    {
        s_iArray[s_iCount++] = i;
        return true;
    }
    else
    {
        return false;
    }
}

template<class T, int size>
bool Stack<T, size>::pop(T &i)
{
    if (s_iCount >= 0)
    {
        i = s_iArray[--s_iCount];
        return true;
    }
    else
    {
        return false;
    }
}

```

定义类模板

构造函数

初始化元素个数

声明成员函数

返回元素个数

栈中的元素用数组来存储

定义成员函数

如果栈还没满

加入元素到数组中

如果栈中有元素

弹出元素



```
int main(int argc, char *argv[])
{
    cout<<"--类模板实例化--"<<endl;
    Stack<int> iv;
    iv.push(1); cout<<"压入1"<<endl;
    iv.push(2); cout<<"压入2"<<endl;
    iv.push(3); cout<<"压入3"<<endl;
    cout << "元素个数:" ;
    cout << iv.get_size() <<endl<<endl;
    cout << "弹出元素:" ;
    int e = 0;
    iv.pop( e );
    cout << e << endl;
    cout << "弹出元素:" ;
    iv.pop( e );
    cout << e << endl;
    cout << "元素个数:" ;
    cout << iv.get_size() << endl;
    return 0;
}
```

用int型和默认的非类型参数值实例化类模板 Stack

依次压入3个整数

输出当前栈中的元素个数

弹出元素

图 13-15 类模板实例

如图 13-15 所示的代码中声明一个类模板 Stack，成员函数 push()和 pop()分别用来向类中添加元素和弹出元素。main()函数对类模板 Stack 进行实例化并创建对象，然后调用成员函数进行栈的相关操作。



## 13.4 类模板的静态成员

与普通类一样，模板也可以有自己的静态成员。不同的是，每种类型的类模板的实例有自己的一组静态成员。与普通类的静态成员初始化一样，类模板的静态成员也需要在类内声明，在类外定义。其在类外定义的一般形式如图 13-16 所示。

```
template <类型名 参数名1, 类型名 参数名2, ...>
类型名 类名<类型名 参数名1, 类型名 参数名2, ...>::静态数据成员= 初始化值;
```

示例

```
template <class T, int size>
class Stack
{
    ...
private:
    static int sNumber ;
    ...
};
```

```
template <class T, int size>
int Stack<T, size>::sNumber;
```

编译器会自动赋初值为 0

图 13-16 在类外定义类模板的静态成员的一般形式





【示例 13-2】下面是坦克计数程序，类模板 CTankManger 用不同类型的 Tank 来实例化，并统计这种类型 tank 的数量，其实现代码及结果如图 13-17 所示。

```
#include <iostream.h>
template<class T>
class CTankManger
{
public:
    bool PushTank(T& tank);
    void OutputTankNumber();
private:
    enum{size = 10};
    static int siCount;
    mTankArray[size];
};

template<class T>
int CTankManger<T>::siCount;

template<class T>
bool CTankManger<T>::PushTank(T& tank)
{
    if(siCount < size)
    {
        cout << "Push tank type :";
        tank.Print();
        mTankArray[siCount] = tank;
        ++siCount;
        return true;
    }
    return false;
}

template<class T>
void CTankManger<T>::OutputTankNumber()
{
    cout << siCount << endl;
}

class CT91
{
public:
    void Print()
    {
        cout << "T91 tank!" << endl;
    }
};

class CT20
{
public:
    void Print()
    {
        cout << "T20 tank!" << endl;
    }
};
```

定义类模板

声明类模板的成员函数

声明类模板的静态成员

定义类模板的静态成员

定义类模板成员函数

定义类模板成员函数

定义类

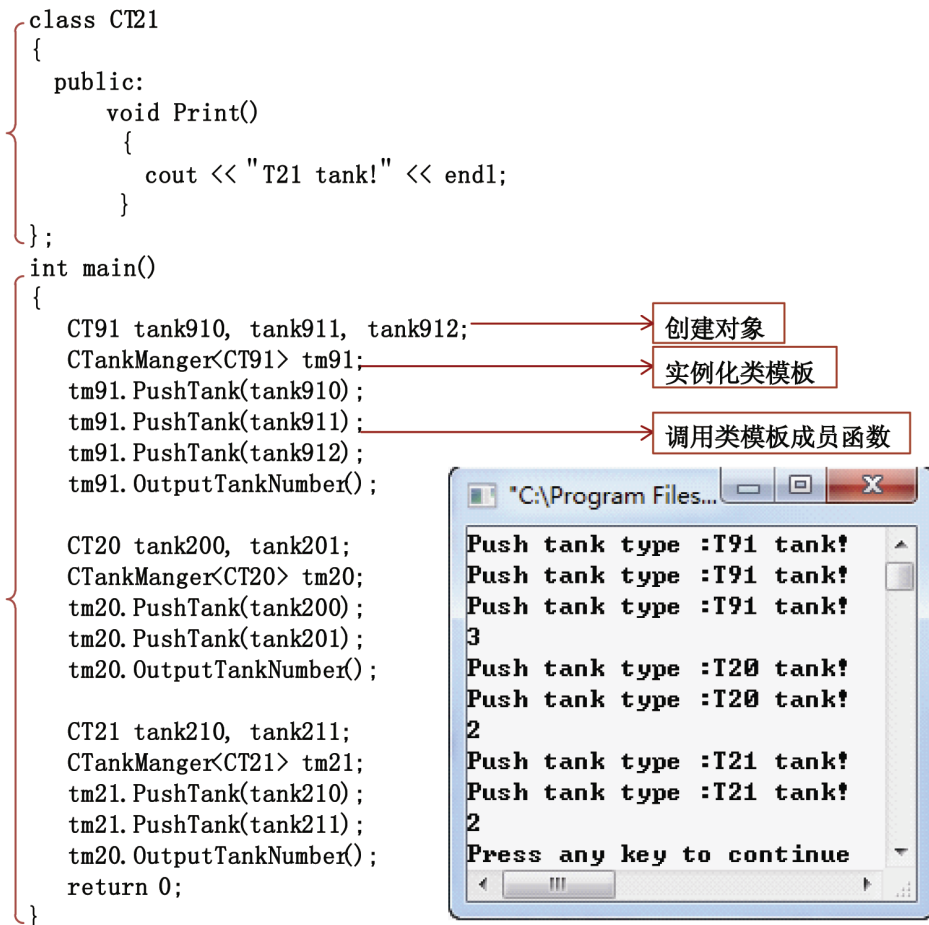


图 13-17 类模板的静态成员实例

程序中的 `CTankManager` 是一个管理类，用来管理放在其中的 `tank`，其中的 `siCount` 是坦克的计数器，每种类型的坦克都有自己的计数器。`CT91`、`CT20`、`CT21` 是具体的坦克类型。



## 13.5 类模板的友元

类模板的友元和普通类的友元一样，使其友元类或友元函数拥有存取的特权，即可以访问到类中的私有或保护的数据成员和成员函数。有 3 种友元可以出现在类模板中，如图 13-18 所示。

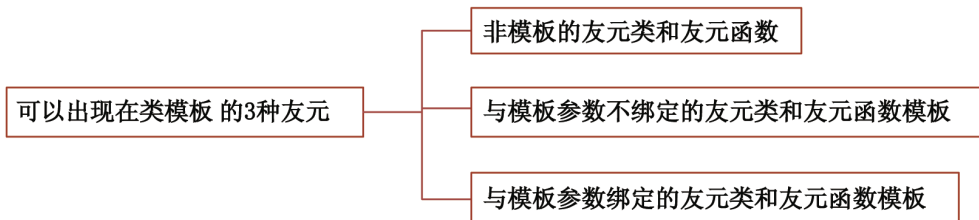


图 13-18 可以出现在类模板中的友元



### 13.5.1 非模板的友元类和友元函数

类模板可以接受一个非模板的类或者函数作为友元。对于这种友元类和友元函数，不需要在声明友元前声明，原因如图 13-19 所示。

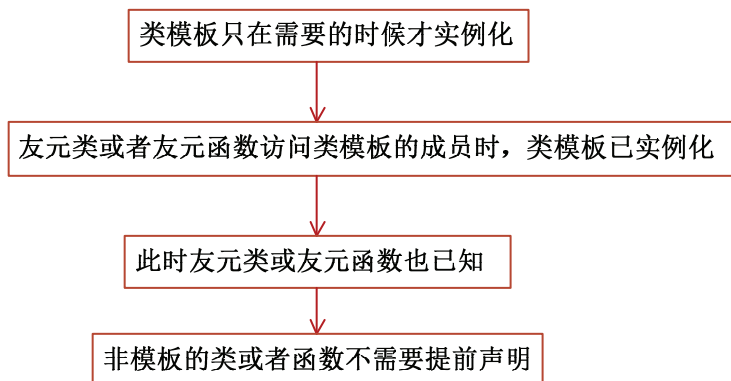


图 13-19 不需要在声明友元前声明友元类和友元函数的原因

但要把类成员函数声明为友元，则前面必须有类的定义（注意不是声明，是定义），因为一个类成员只能由该类的定义引入。例如，如图 13-20 所示的代码是在类模板中声明一个友元类和友元函数。

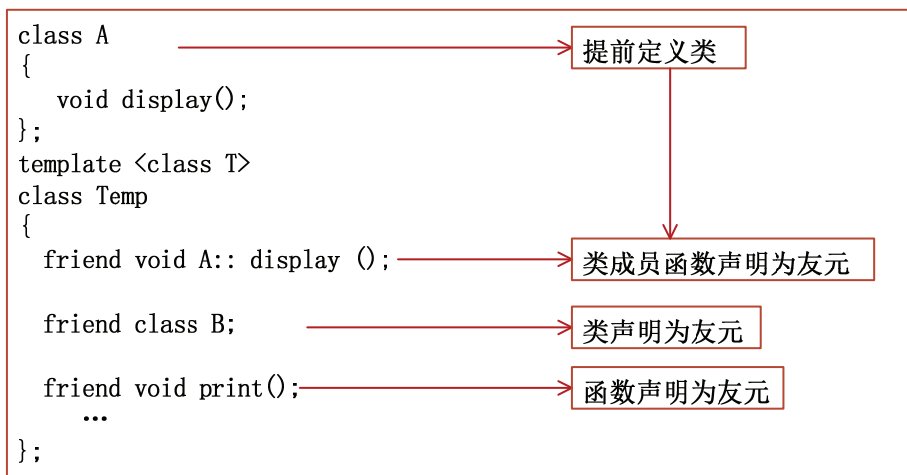


图 13-20 类模板中声明一个友元类和友元函数

### 13.5.2 与模板参数不绑定的友元类和友元函数模板

类模板的友元可以是类模板或函数模板的实例。如果后者的模板实参与前者的模板参数没有任何关系，也就是说友元模板与类模板的参数不绑定，则在声明友元前不需要提前声明。其原因同使用非模板的友元一样。

例如，如图 13-21 所示的代码是在类模板中声明与模板参数不绑定的友元类和友元函数模板。



声明与模板参数不绑定的友元类和友元函数模板

```
template <class Type >
class Temp
{
    template <class T >
    friend void A <T>:: display ();

    template <class T >
    friend class B ;

    template <class T >
    friend void print (Temp <T>);
    ...
};
```

图 13-21 类模板中声明与模板参数不绑定的友元类和友元函数模板

### 13.5.3 与模板参数绑定的友元类和友元函数模板

如果类模板的友元是类模板或者函数模板的实例，而且其模板实参就是类模板的参数，则友元模板必须提前声明。

例如，如图 13-22 所示的代码，是在类模板中声明与模板参数绑定的友元类和友元函数模板。

声明与模板参数绑定的友元类和友元函数模板

```
template <class Type>
class B
{ ...
};

template <class Type >
void print (Temp <Type>);

template <class Type >
class A
{
    void display ();
    ...
};

template <class Type >
class Temp
{
    friend void A <Type> ::display ();
    friend class B <Type> ;
    friend void print <Type> (Temp <Type>);
    ...
};
```

提前声明或定义

声明类模板的友元语句中的<Type>不能少

图 13-22 类模板中声明与模板参数绑定的友元类和友元函数模板



## 13.6 类模板的特化

有时某些类型不能直接用来实例化类模板，或者说直接实例化不能满足需要，此时就要针对这种类型进行特化，包括完全特化和偏特化。

### 13.6.1 类模板的全特化

类模板的特化是针对特殊的类型，进行特殊的处理。例如，图 13-23 中为 Stack 类模板添加返回最大值的 max() 函数。

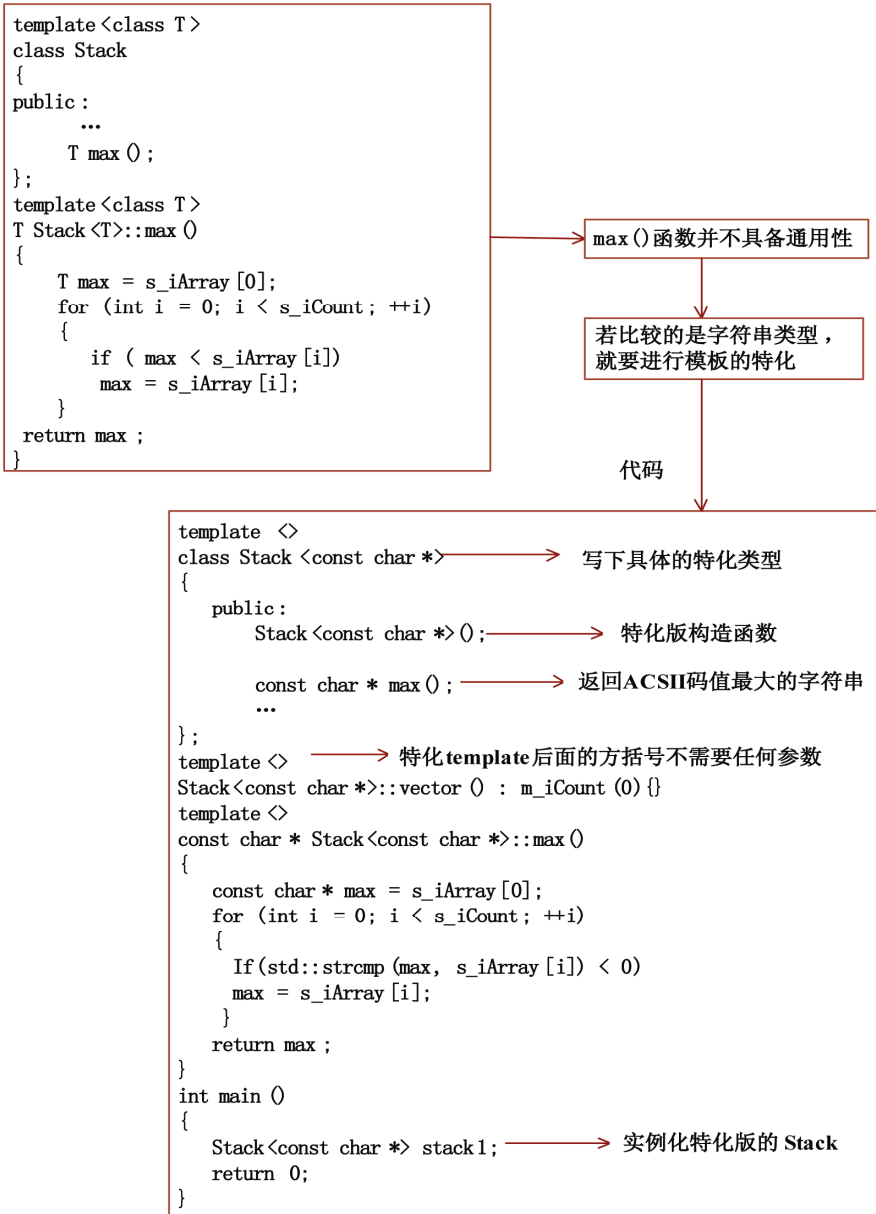


图 13-23 类模板的全特化



### 13.6.2 类模板的偏特化

类模板的偏特化是指需要根据模板的某些参数但不是全部的参数进行特化。

例如，一个 Dot 类，其模板参数列中有两个参数 x 和 y。假如要求对在纵坐标等于 80 的点进行一些特别处理。这时就需要对模板参数的 y 进行常量量化，如图 13-24 所示。

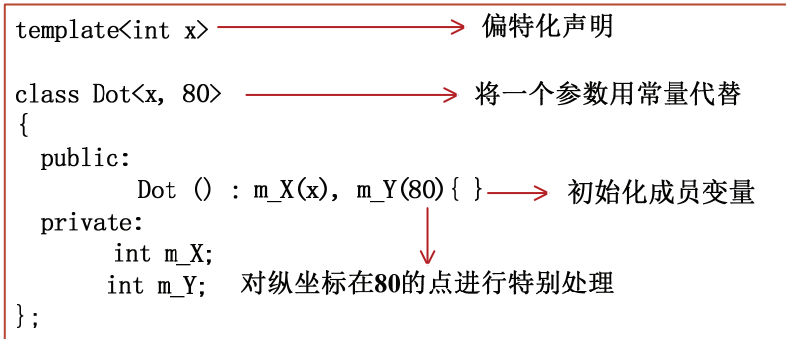


图 13-24 模板参数的 y 进行常量量化

部分特化的模板实参表只列出那些模板实参仍然未知的参数。如图 13-25 所示是一个偏特化 Stack 类的示例。

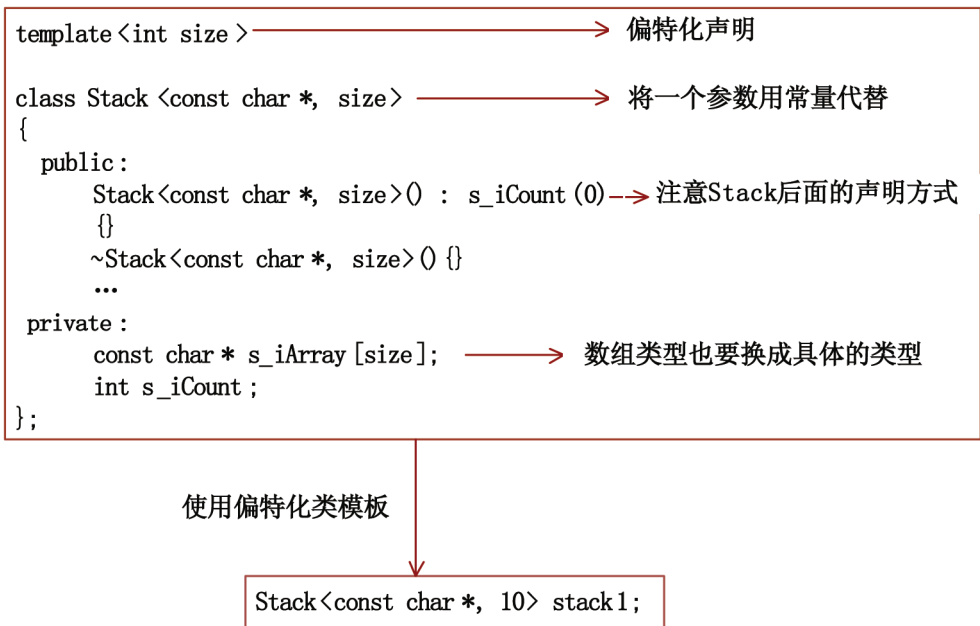


图 13-25 类模板的偏特化

### 13.6.3 类模板的匹配规则

类模板的匹配规则遵循“特化程度最高”的模板。具体来说，模板参数最准确匹配的拥有最高的实例化优先权（即被编译器优先选择来实例化），如图 13-26 所示的几种实例化。



<code>template &lt;class T&gt; class Stack{};</code>	→ 普通型
<code>template &lt;class T&gt; class Stack&lt;T*&gt;{};</code>	→ 对指针类型特化
<code>template&lt;class T&gt; class Stack&lt;const T*&gt;{};</code>	→ 对const 型指针特化
<code>template &lt;&gt; class Stack &lt;void*&gt;{};</code>	→ 对void*进行特化

根据Stack不同的模板参数，会寻找最匹配的版本进行实例化

如果是指针类型会选择第二行定义的模板来实例化；  
如果一个类型的指针是空类型的，就会选择第四行的进行实例化；  
常量类型的指针会选择第三行的模板。

图 13-26 类模板的匹配规则



## 13.7 小结

本章主要讲述了类模板的概念及类模板的定义和实例化。C++中的类模板与普通类相似，可以定义自己的函数成员、静态成员、友元函数等。类模板有全特化和偏特化两种，要注意在此之前需要普通类模板的定义。



## 13.8 习题

【题目 13-1】使用类模板编程求整数  $n$  的阶乘  $n!$ 。程序运行结果如图 13-27 所示。

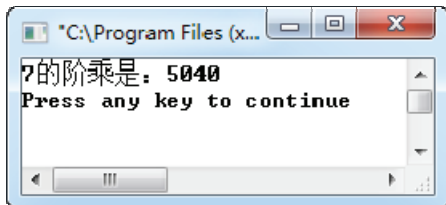


图 13-27 运行结果

【题目分析】本题主要考查类模板的知识。

【关键代码】

```
#include <iostream.h>
template <int x>
class Factorial
{
public:
    enum { Result =Factorial<x-1>::Result*x };
};
```



```
template <>
class Factorial<1>
{
    public:
        enum { Result=1 };
};

int main()
{
    cout<<"7 的阶乘是: "<<Factorial<7>::Result <<endl;
    return 0;
}
```



## 第 4 篇 C++应用技术篇

# 第 14 章 输入/输出流

应用程序的输入/输出使用非常频繁。一般而言，输入是为了实现程序与用户的交流，而输出是为了返回结果或给出提示信息。本章将详细介绍 C++中输入/输出流的有关知识。

## 14.1 输入/输出流的引入

如果一个应用程序没有输入和输出，那它也就没有应用价值。在 C++中，输入/输出功能是通过调用该操作系统的 I/O 库来实现的。

### 14.1.1 C 语言中的输入/输出缺陷

C 语言中的输入/输出大都是由函数 scanf()和 printf()来实现的,具体实现过程如图 14-1 所示。

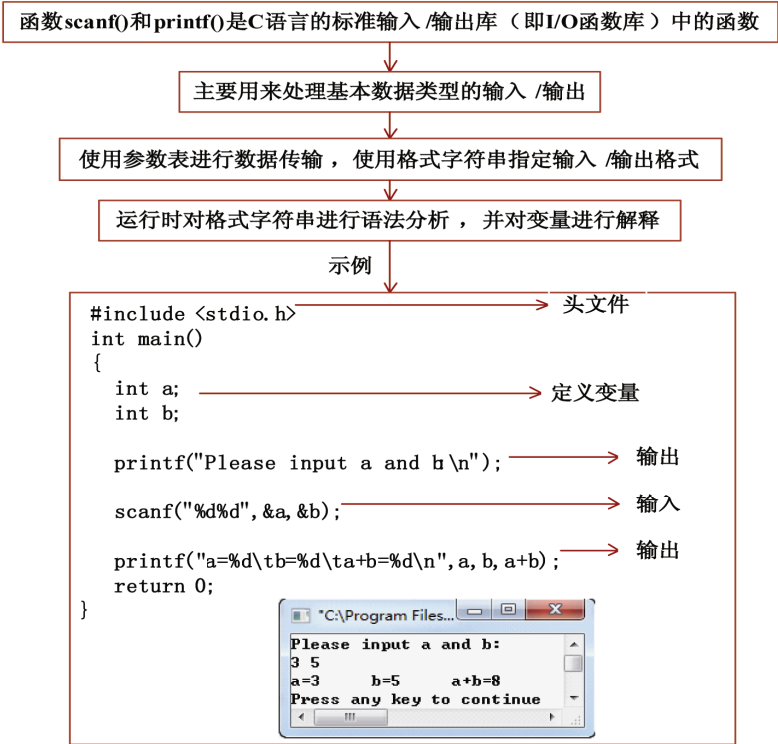


图 14-1 函数 scanf()和 printf()



在如图 14-1 所示的代码中，使用 C 语言的标准 I/O 库函数 `scanf()` 和 `printf()` 来实现程序的输入/输出，必须添加头文件 `stdio.h`，否则编译将无法通过。

既然使用 C 语言 I/O 库函数也能够很好地完成程序的输入/输出，为什么还要引入 C++ 的输入/输出流呢？这是因为使用 C 语言 I/O 库函数存在几个缺陷，如图 14-2 所示。

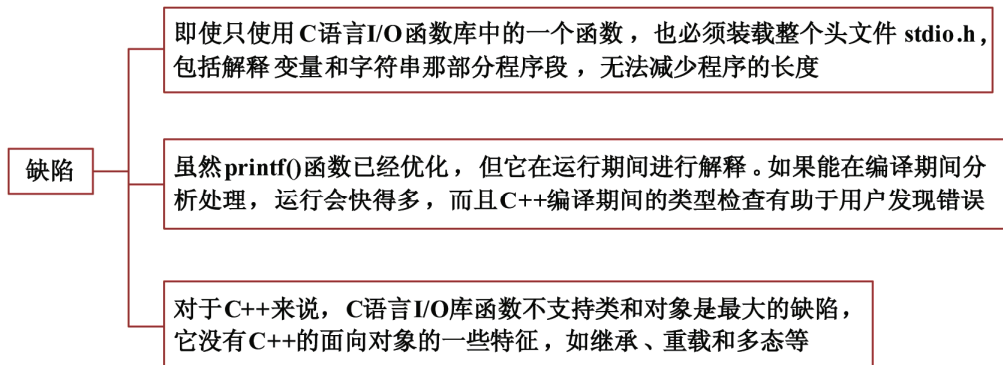


图 14-2 C 语言 I/O 库函数存在的缺陷

### 14.1.2 输入/输出流简介

C++ 除支持 C 的输入/输出系统外，还提供了自己的输入/输出系统，并通过重载运算符“<<”和“>>”来支持类和对象的输入/输出。C++ 的输入/输出系统是以字节流的形式实现的。

C++ 中的流是指数据从一个对象传递到另一个对象的操作，如图 14-3 所示。

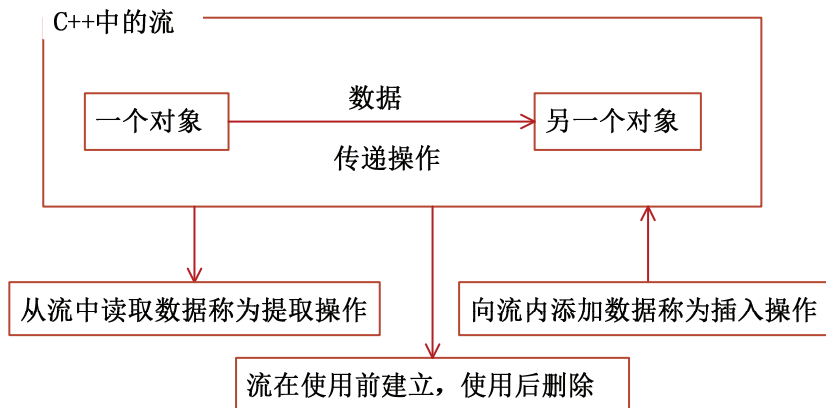


图 14-3 C++ 中的流

如果数据的传递是在设备之间进行的，这种流就称为 I/O 流。C++ 专门内置了一些供用户使用的类，在这些类中封装了可以实现输入/输出操作的函数，这些类统称为 I/O 流类。

流具有方向性，与输入设备相关联的流称为输入流，与输出设备相关联的流称为输出流，统称为输入/输出流，如图 14-4 所示。

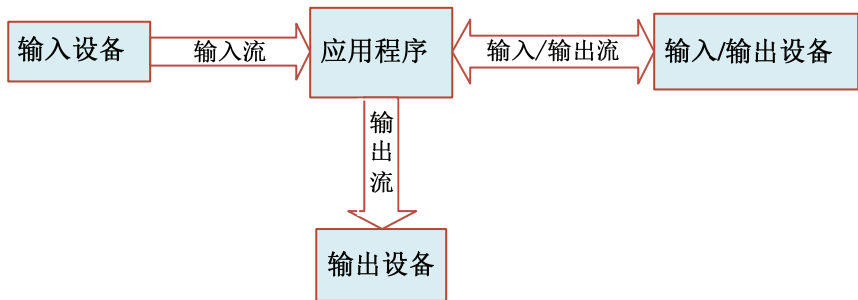


图 14-4 输入/输出流

C++没有使用 C 语言的输入/输出函数库，而是使用 `iostream` 类库。`iostream` 是通过类的继承，以及类的成员函数的重载来实现的。`iostream` 类库使用统一的函数接口操作标准 I/O、文件、存储块等输入/输出设备。同时，`iostream` 类库具有很好的扩展性，用户可通过重载对其进行扩展。

### 14.1.3 输入/输出流类层次

C++的输入/输出流类库是用派生方法建立起来的，它有两个平行的基类，即 `streambuf` 和 `ios`。其他的流类都是从这两个基类直接或间接派生的。

#### 1. `streambuf` 类

`streambuf` 类是带有缓冲区的流类库，它的作用是提供物理设备的接口、缓冲区或处理流的通用方式。当其用做流类库中的基类时，派生出 3 个流类，如图 14-5 所示。

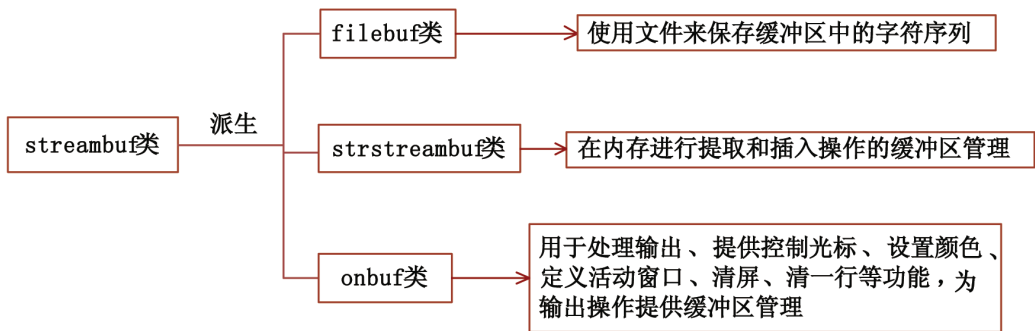


图 14-5 `streambuf` 类的派生类

`streambuf` 类使用的缓冲区由一个字符序列和输入/输出缓冲区指针组成，指针指向字符被取出或插入的位置。通常情况下，均使用图 14-5 中的 3 个派生类，很少直接使用 `streambuf` 类。

#### 2. `ios` 类

`ios` 类及其派生类为用户提供了使用流类的接口，它们均有一个指针指向 `streambuf` 类。`ios` 类及其派生类使用 `streambuf` 来完成对错误的格式化输入/输出的检查，并且支持对 `streambuf` 类的缓冲区进行 I/O 时的格式化或非格式化转换。



ios 作为流类库中的基类，可以派生出许多类，这些类在实际的程序应用中非常频繁。其类的层次关系如图 14-6 所示。

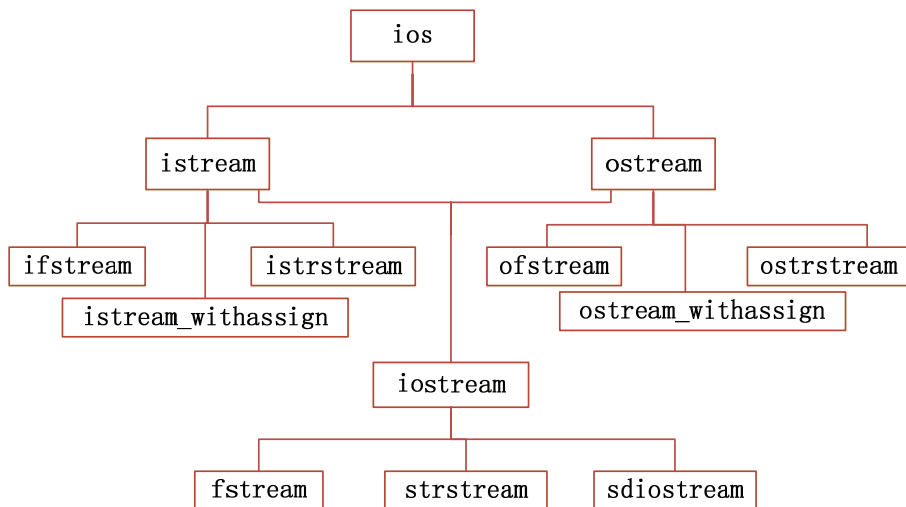


图 14-6 ios 类及其派生类的层次关系

图 14-6 中的 ios、istream、ostream 和带缓冲区的流类库 streambuf 类构成了 C++ 中 iostream 输入/输出功能的基础。

流是一个抽象的概念，实际进行 I/O 操作时，必须将流与一种具体的物理设备联系起来。例如，将流和键盘联系起来，当从该流中提取数据时，就是从键盘输入数据。用户也可以用 istream、ostream 等类声明流对象，形式如图 14-7 所示。



图 14-7 用 istream、ostream 等类声明流对象

使用流类库的程序，不但可以识别标准的 I/O 设备，还可以重载运算符“<<”和“>>”，使程序识别用户自定义的类型，从而极大地提高了程序的可靠性和灵活性。



## 14.2 标准输入/输出流

C++ 将一些常用的流类对象（如键盘输入、显示器输出、程序运行出错输出、打印机输出等）定义并内置在系统中，供用户直接使用。这些系统内置的用于设备间传递数据的对象称为标准流类对象，共有 4 个，如图 14-8 所示。在一些系统中，不需要关闭文件，当程序结束时，文件会自动关闭。但是，关闭文件是一个好的习惯。而且，如果要用相同的文件流变量打开另外的文件，必须先关闭用文件流变量打开的文件。

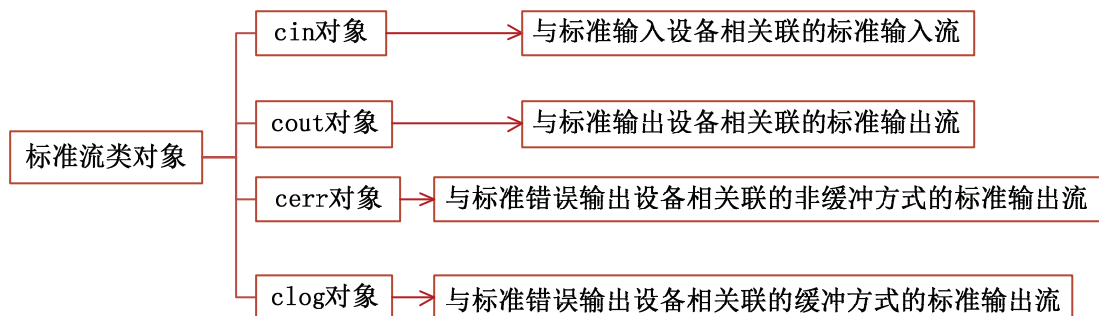


图 14-8 4 个标准流类对象

本节只介绍 cin 和 cout 对象的使用，有关 cerr 和 clog 对象的使用不做讲解。

### 14.2.1 标准输出流对象

标准输出流对象是采用 cout 对象将输出流中的数据显示在屏幕上，称为输出操作。使用 cout 对象时要结合流输出运算符“<<”，也叫流插入运算符。

【示例 14-1】下面的程序通过引用标准输出流对象输出了一个简单的字符串。其实现代码及结果如图 14-9 所示。

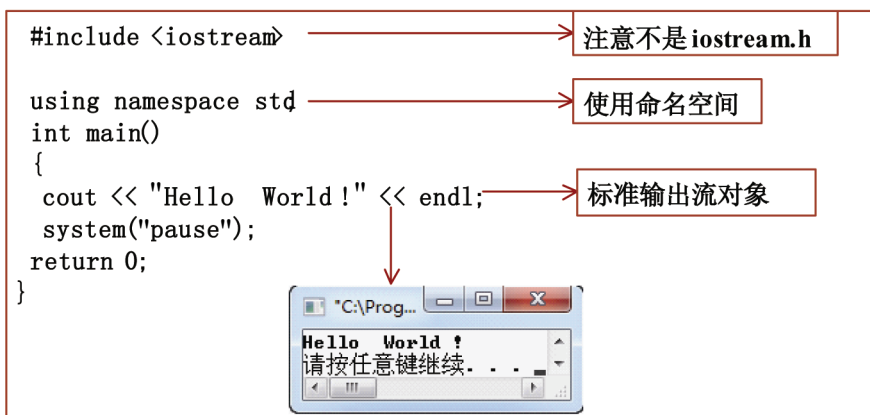


图 14-9 标准输出流对象使用实例



**注意：**（1）<iostream>和<iostream.h>是不一样的，命名空间在后面详细介绍。  
（2）重载的运算符“<<”和“>>”都可以在一条语句中连续使用。

### 14.2.2 标准输入流对象

与标准输出流对象相对应，标准输入流用于将键盘等输入设备中的信息传送到程序中，其使用的是 cin 对象，称为输入操作。使用 cout 对象时要结合流输入运算符“>>”，也叫流读取运算符。

【示例 14-2】下面的程序段根据输出提示，接收从键盘输入的两个整型变量的值，并将输入的变量值进行简单的求和运算后将结果输出。其实现代码及结果如图 14-10 所示。

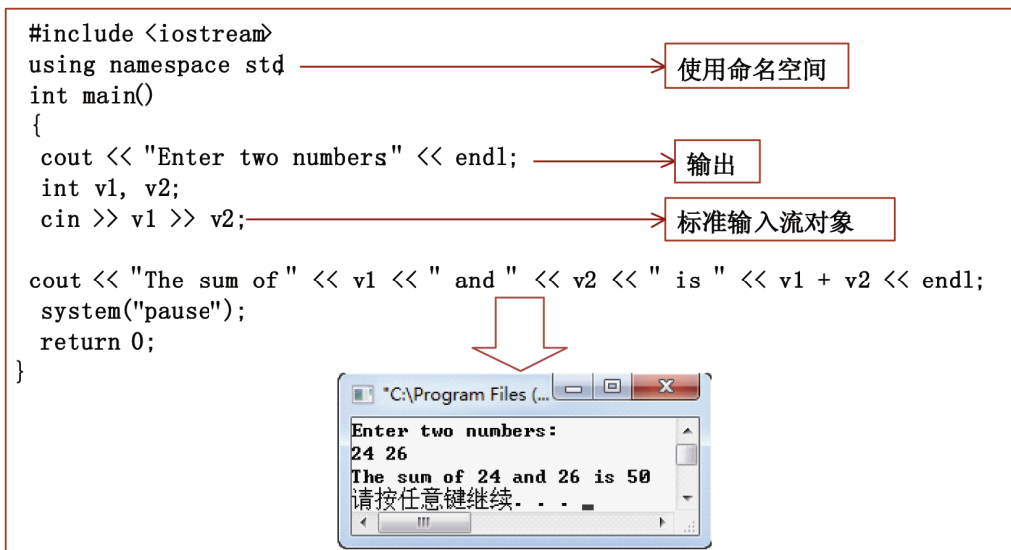


图 14-10 标准输入流对象使用实例



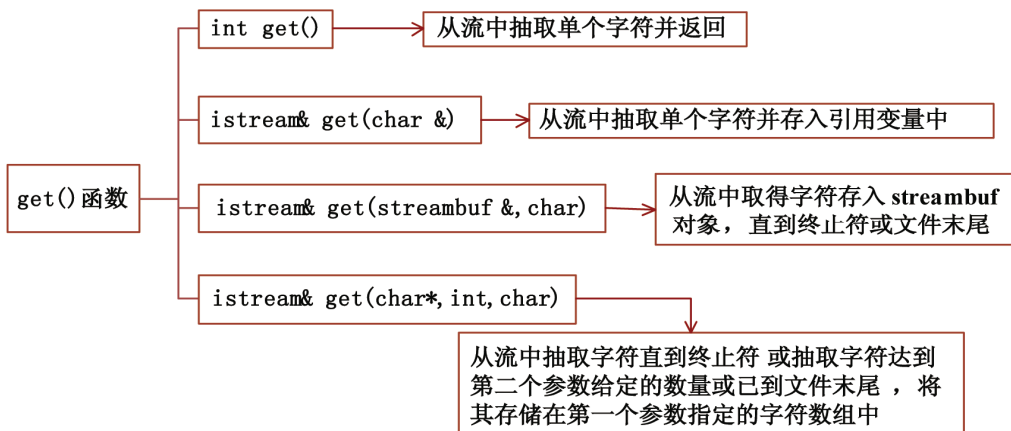
**提示：**在复杂的程序表达式中要注意重载运算符“<<”和“>>”的优先。

## 14.3 输入/输出流成员函数

在 C++ 中，输入/输出流除了可以使用前面介绍的输入/输出流对象外，类 `istream` 和类 `ostream` 还有几个从流中进行非格式化抽取的成员函数：`get()`、`getline()`、`put()`、`read()` 和 `write()` 函数及其他成员函数。

### 14.3.1 `get()` 函数

根据 `get()` 函数具体应用的范围不同，其通过函数重载可以实现不同的功能，如图 14-11 所示。

图 14-11 `get()` 函数



【示例 14-3】下面的程序实现 `get()` 函数的几种不同用法，其分别根据用户从键盘上的输入来输出一个字符、一个字符串等。其实现代码及结果如图 14-12 所示。

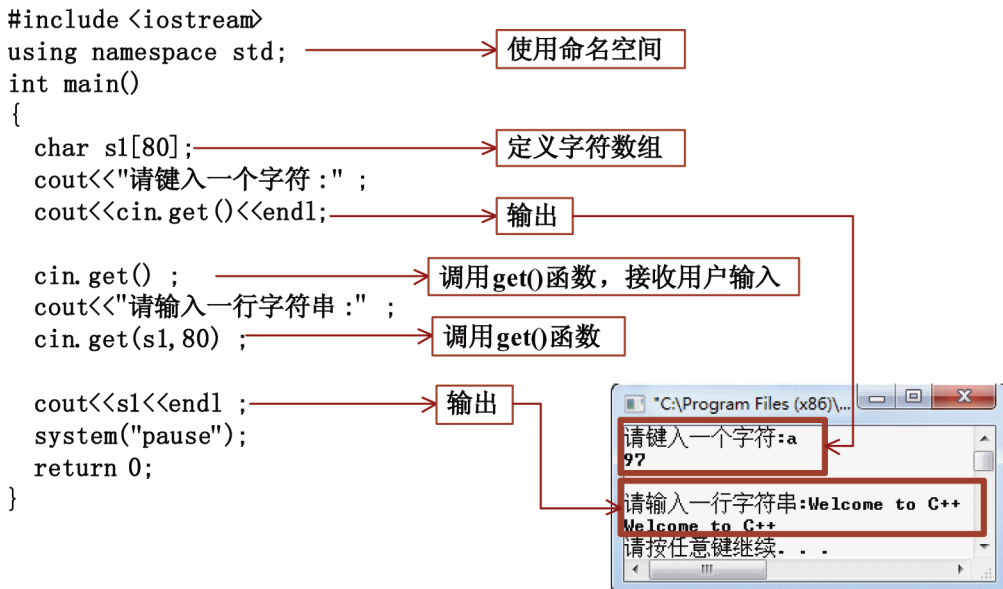


图 14-12 `get()` 函数实例

`get()` 函数有许多形式，其重载后的形式可包含参数，返回类型也可为字符串，但在实际程序中一般用不带参数的形式。

### 14.3.2 `getline()` 函数

`getline()` 函数可以实现一个字符串的获取，其定义的一般格式如图 14-13 所示。

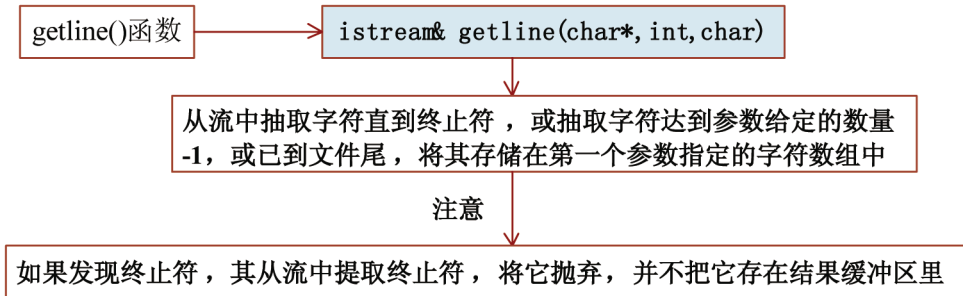


图 14-13 `getline()` 函数定义的一般形式

【示例 14-4】下面的程序通过 `getline()` 函数获取字符串。其实现代码及结果如图 14-14 所示。



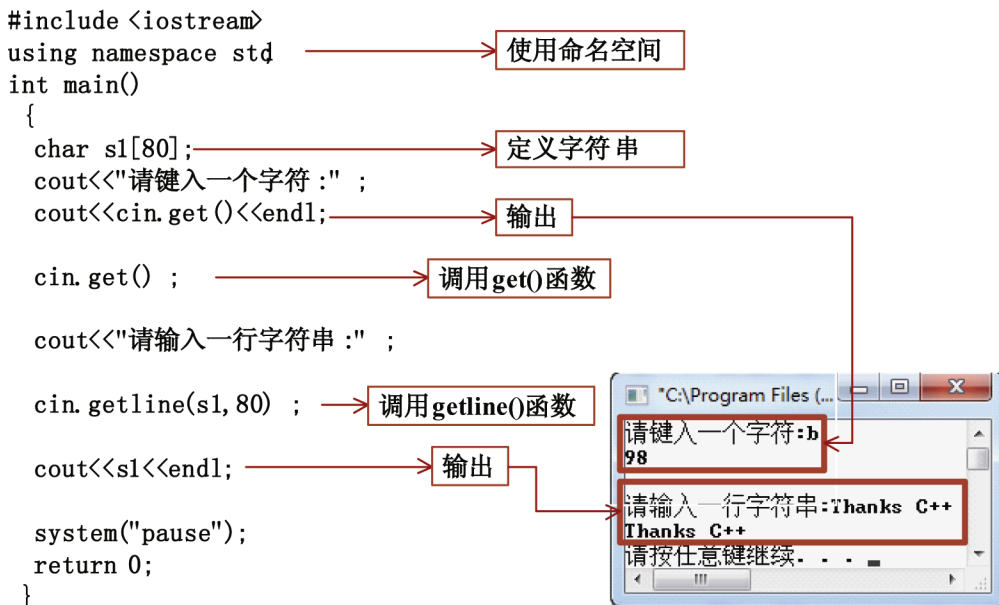


图 14-14 getline()函数实例

get()和 getline()都可以给参数赋值,在输入字符和字符串时,C++会有一个结束字符 "\0"。用 get()函数输入时,会在参数中留下这个字符,而用 getline()函数则不会留下,这是两者的主要区别。

### 14.3.3 put()函数

put()函数用于输出一个字符,如图 14-15 所示。



图 14-15 put()函数输出一个字符

和 "<<" 一样,一条语句中也可以连续调用 put()函数,如图 14-16 所示。



图 14-16 连续调用 put()函数

put()函数返回调用 put 的对象的引用。还可以用 ASCII 码值表达式调用 put()函数,如语句 cout.put(65)也输出字符'A'。

### 14.3.4 read()和 write()函数

调用成员函数 read()、write()可实现无格式输入/输出,如图 14-17 所示。

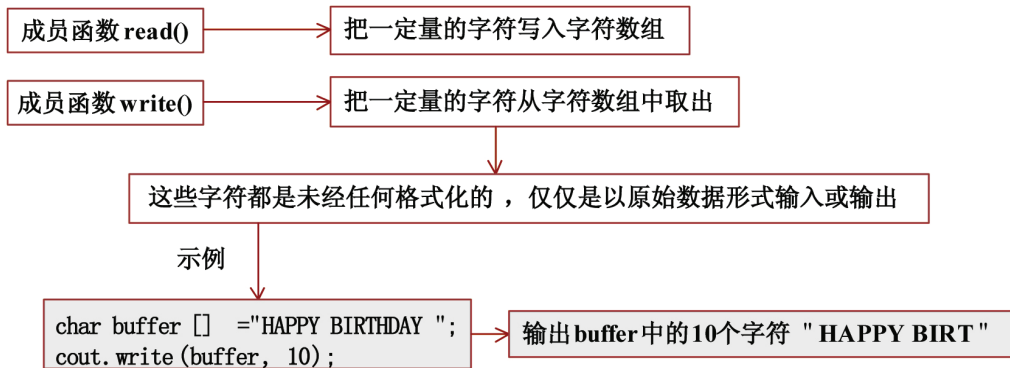


图 14-17 成员函数 read()和 write()

成员函数 read()把指定个数的字符输入到字符数组中。如果读取的字符个数少于指定的数目，可以设置标志位 failbit。成员函数 gcount()统计最后输入的字符个数。

【示例 14-5】下面的程序演示了类 istream 中的成员函数 read()和 gcount()，以及类 ostream 中的成员函数 write()的使用。其实现代码及结果如图 14-18 所示。

```
#include <iostream>
using namespace std;
int main()
{
    const int SIZE = 80;
    char buffer[SIZE];

    cout << "Enter a sentence\n";
    cin.read( buffer, 20 );

    cout << "\nThe sentence entered was\n";
    cout.write( buffer, cin.gcount() );

    cout << endl;
    system("pause");
    return 0;
}
```

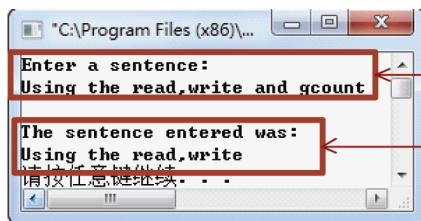


图 14-18 成员函数 read()和 write()实例

### 14.3.5 其他成员函数

istream 类和 ostream 类中的其他成员函数包括 ignore()、putback()和 peek()等，如图 14-19 所示。

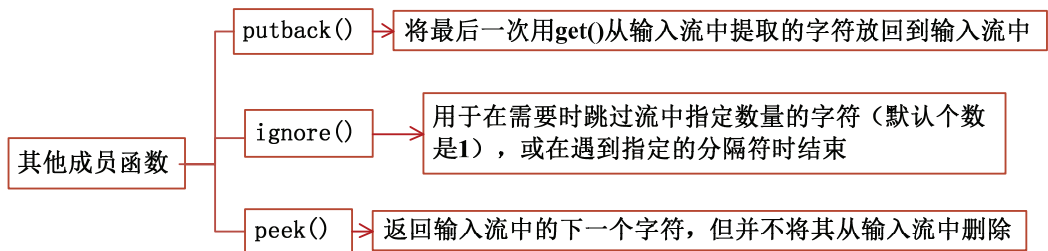


图 14-19 其他成员函数



## 14.4 输入/输出格式控制

C++使用C中的printf()和scanf()函数进行格式化控制的同时又提供了两种格式化控制的方法，如图 14-20 所示。



图 14-20 两种格式化控制的方法

### 14.4.1 用 ios 类的成员函数进行格式控制

一般来说，ios 类的成员函数进行格式控制主要是通过对格式状态字、域宽、填充字符和输出精度操作来完成的。

#### 1. 状态字

状态字也称状态标志字，其类型为 long int。它是在 ios 类的 public 部分定义了一个枚举，此枚举类型的每个成员分别定义状态字的一个位，每个位称为状态标志位，该枚举的定义如图 14-21 所示。

状态字	二进制表示	说明
enum		
{		
skipws	0x0001	跳过输入中的空白，用于输入
left	0x0002	左对齐输出，用于输出
right	0x0004	右对齐输出，用于输出
internal	0x0008	在符号位和基指示符后填入字符，用于输出
dec	0x0010	转换基数为十进制，用于输入或输出
oct	0x0020	转换基数为八进制，用于输入或输出
hex	0x0040	转换基数为十六进制，用于输入或输出
showbase	0x0080	输出时显示基指示符，用于输入或输出
showpoint	0x0100	输出时显示小数点，用于输出
uppercase	0x0200	十六进制输出时，表示制式的和表示数值的一律为大写，用于输出
showpos	0x0400	正整数前显示“+”符号，用于输出
scientific	0x0800	用科学表示法显示浮点数，用于输出
fixed	0x1000	用定点形式显示浮点数，用于输出
unitbuf	0x2000	在输出操作后立即刷新所有流，用于输出
stdio	0x4000	在输出操作后刷新stdout和stderr，用于输出
}		

图 14-21 状态字



这些枚举元素的值的共同特点是，使状态标志字二进制表示中的不同位为 1，使它们共同组成状态标志字存放在数据成员 `long x_flags` 中。这些状态之间是或的关系，可以几个并存。

## 2. 用 ios 类的成员函数进行格式控制

ios 类提供了几个用于控制输入/输出格式的成员函数，其中的参数 `flags` 是状态控制字，存放在 ios 类中的数据成员 `long x_flags` 中。主要成员函数的使用方法如下。

设置状态标志，使用函数 `setf()` 将状态标志位置“1”，函数原型如图 14-22 所示。

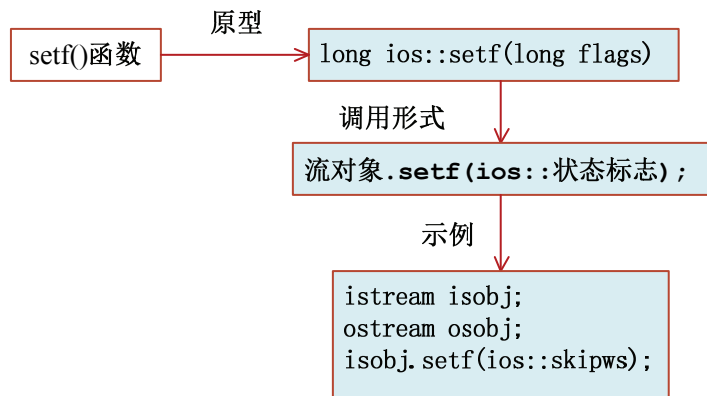


图 14-22 setf()函数原型

清除状态标志指用函数 `unsetf()` 将某一状态标志置“0”，函数原型如图 14-23 所示。

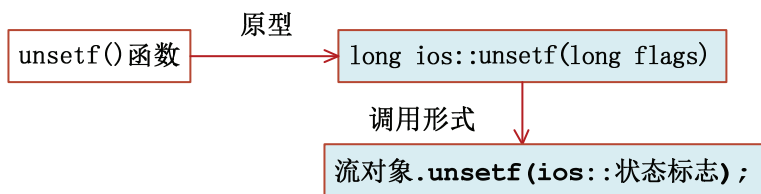


图 14-23 unsetf()函数原型

取状态标志指用函数 `flags()` 返回状态标志字，有带参数和不带参数两种形式，它们的函数原型如图 14-24 所示。

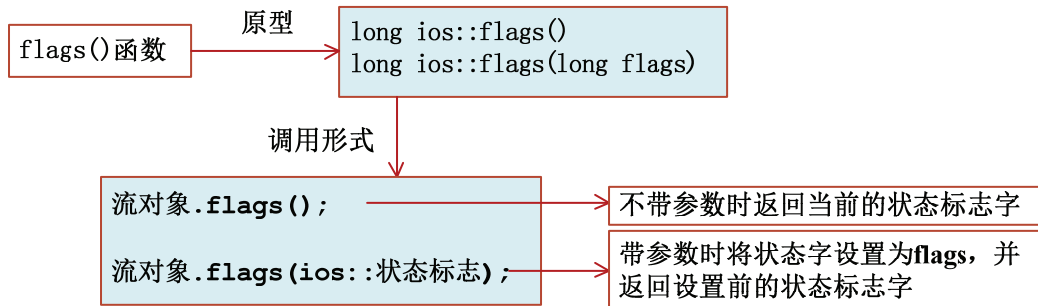


图 14-24 flags()函数原型

`flgs()` 与 `setf()` 的区别如图 14-25 所示。

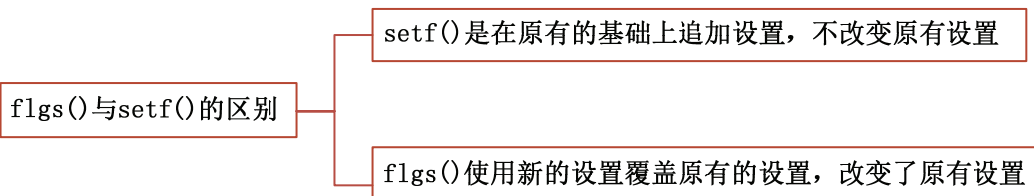


图 14-25 flgs()与 setf()的区别

【示例 14-6】下面的程序使用成员函数控制输入/输出格式。程序中输出多个字符和字符串，其中使用多种 cout 对象的成员函数来控制格式。其实现代码及结果如图 14-26 所示。

```

#include <iostream>
using namespace std;
int main()
{
    cout<<"x_width="<<cout.width()<<endl;
    cout<<"x_fill="<<cout.fill()<<endl;
    cout<<"x_precision="<<cout.precision()<<endl;
    cout<<123<<"    "<<123.456789<<endl;
    cout<<"-----"<<endl;
    cout.width(10);
    cout.precision(3);
    cout<<123<<"    "<<123.456789<<endl;
    cout<<"x_width="<<cout.width()<<endl;
    cout<<"x_fill="<<cout.fill()<<endl;
    cout<<"x_precision="<<cout.precision()<<endl;
    cout<<"-----"<<endl;
    cout.width(10);
    cout.fill(' *');
    cout<<123<<"    "<<123.456789<<endl;
    cout.width(10);
    cout.setf(ios::left);
    cout<<123<<"    "<<123.456789<<endl;
    cout<<"x_width="<<cout.width()<<endl;
    cout<<"x_fill="<<cout.fill()<<endl;
    cout<<"x_precision="<<cout.precision()<<endl;
    system("pause");
    return 0;
}
  
```

设置默认域宽、填充默认字符空格、设置默认精度

设置域宽 10、设置精度 3

调用 width() 函数、调用 fill() 函数、调用 precision() 函数

设置填充 " \* "

设置状态标志

图 14-26 使用成员函数控制输入/输出格式实例

使用成员函数可以按照用户自己的需求对输出的格式进行控制，其主要包括对输出域宽、精度和填充字符等方面的设置。



14.4.2 使用格式控制符进行格式控制

使用 ios 类的成员函数控制输入/输出格式时，每个函数的调用都要写一条语句，它们不能直接嵌入到输入/输出语句中，使用很不方便。因此，C++提供了另外一种输入/输出格式的控制方法——使用一种称为格式控制符的特殊函数。

1. 预定义的格式控制符

预定义的格式控制符可以直接嵌入到输入/输出语句中，完成类似于 ios 类中控制输入/输出格式的成员函数的功能。一般来说，C++的预定义格式控制符如表 14-1 所示。

表 14-1 预定义的格式控制符

格式控制符	功 能
dec	以十进制形式输入/输出整型数，用于输入或输出
hex	以十六进制形式输入/输出整型数，用于输入或输出
oct	以八进制形式输入/输出整型数，用于输入或输出
ws	用于输入时跳过开头的空白符，仅用于输入
endl	插入一个换行符并刷新输出流，仅用于输出
ends	插入一个空字符，用来结束一个字符串，仅用于输出
flush	刷新一个输出流，仅用于输出
setbase(int n)	把转换基数设置位 n(n=0,8,10,16)，默认值为 0（十进制）
resetiosflags(long f)	关闭由参数 f 指定的格式标志，用于输入或输出
setiosflags(long f)	设置由参数 f 指定的格式标志，用于输入或输出
setfill(int c)	设置 c 为填充字符，默认为空格，用于输入或输出
setprecision(int n)	设置小数位数，默认为 6 位，用于输入或输出
setw(int n)	设置域宽，用于输入或输出

其中，格式控制符 setiosflags(long f)和 resetiosflags(long f)中的格式标志与 ios 类中控制输入/输出格式的成员函数所用的标志基本相同。

2. 预定义格式控制符的使用

预定义格式控制符分为带参数和不带参数的两种，如图 14-27 所示。

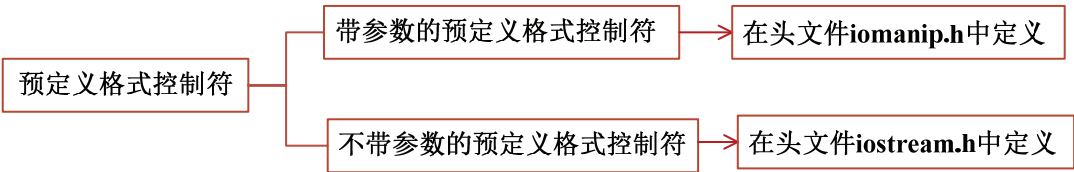


图 14-27 预定义格式控制符的分类

在使用预定义格式控制符时，程序中应包含相应的头文件。格式控制符被嵌入到输入/输出语句中控制输入/输出的格式，而不是执行输入/输出操作。

【示例 14-7】下面的程序使用预定义的格式控制符控制输入/输出格式。其实现代码及结果如图 14-28 所示。



```
#include<iomanip>
#include<iostream>
using namespace std;
int main()
{
    cout<<setw(10)<<987<<654<<endl;
    cout<<987<<setiosflags(ios::scientific)<<setw(15)<<987.654321<<endl;

    cout<<987<<setw(10)<<hex<<987<<endl;
    cout<<987<<setw(10)<<oct<<987<<endl;
    cout<<987<<setw(10)<<dec<<987<<endl;

    cout<<resetiosflags(ios::scientific)<<setprecision(3)<<987.654321<<endl;
    cout<<setiosflags(ios::left)<<setfill('&' )<<setw(7)<<987<<endl;
    cout<<setiosflags(ios::right)<<setfill('#' )<<setw(7)<<987<<endl;
    system("pause");
    return 0;
}
```

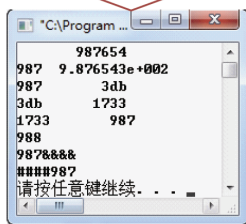


图 14-28 预定义格式控制符实例

如图 14-28 所示的代码中使用了 `setw` 控制符设置域宽、`setiosflags` 控制符设置标识位等，分别输出了一个数的十进制、十六进制和八进制的值，此外对一些输出使用 `setfill` 控制符进行了字符填充的操作。



**注意：**格式控制符 `setw` 只对最靠近它的输出起作用，格式控制符 `dec`、`oct` 和 `hex` 的作用则一直保持到重新设置为止，格式控制符 `setprecision` 在输出时做四舍五入处理。

### 3. 自定义的格式控制符

除了上述系统定义的格式控制符外，用户还可自定义格式控制符。

在 C++ 中，输出流与输入流分别自定义格式控制符的一般形式，如图 14-29 所示。

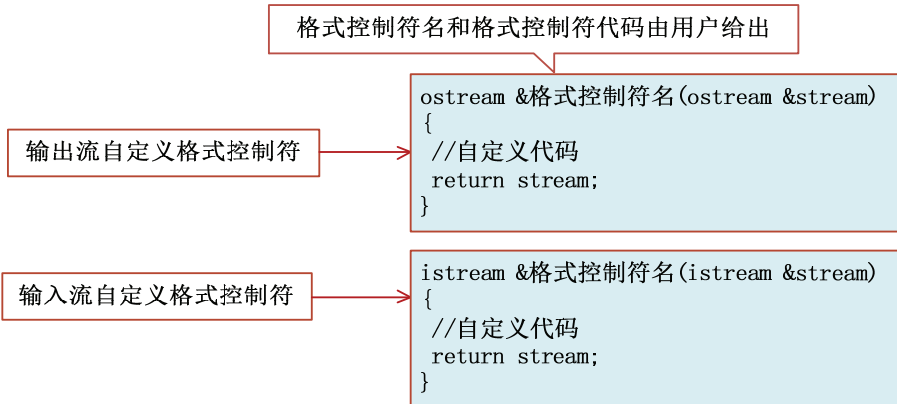


图 14-29 自定义的格式控制符



**提示：**不要丢了返回语句，它是自定义格式控制符的关键。

【示例 14-8】下面的程序使用自定义格式控制符控制输出格式。其实现代码及结果如图 14-30 所示。

```
#include<iomanip>
#include <iostream>
using namespace std;
ostream &outputl(ostream &out)
{
    cout.setf(ios::left);
    cout<<setw(10)<<dec<<setfill(' * ');
    return out;
}
int main()
{
    cout<<345<<endl;
    cout<<outputl<<345<<endl;
    system("pause");
    return 0;
}
```

包含头文件

定义格式

从左输出

域宽为10，十进制输出，填充\*

输出默认格式的字符串

自定义格式控制符输出

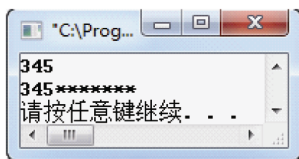


图 14-30 自定义格式控制符实例



## 14.5 用户自定义数据类型的输入/输出

用户自定义数据类型的输入/输出，是通过重载运算符“<<”和“>>”实现的。

### 14.5.1 重载输出运算符“<<”

重载输出运算符“<<”也称为插入运算符，用于用户自定义类型数据的输出。定义重载运算符“<<”的一般形式，如图 14-31 所示。

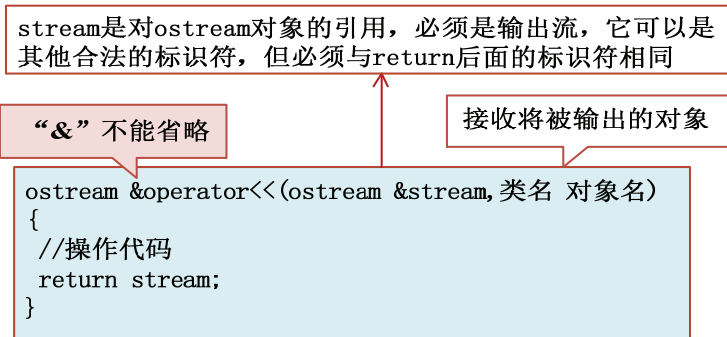


图 14-31 重载运算符“<<”的一般形式





在使用重载输出运算符“<<”时，要注意两个问题，如图 14-32 所示。

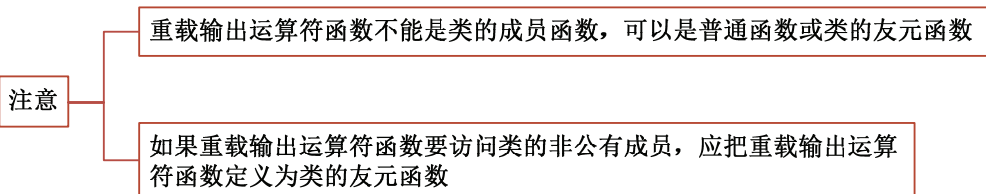


图 14-32 使用重载输出运算符“<<”时，要注意的问题

【示例 14-9】下面的程序将输出运算符“<<”重载为友元函数，实现输出类中数据成员的值。其实现代码及结果如图 14-33 所示。



图 14-33 重载输出运算符“<<”实例

### 14.5.2 重载输入运算符“>>”

重载输入运算符“>>”也称为提取运算符，用于用户自定义类型数据的输入。定义运算符“>>”重载函数的一般形式如图 14-34 所示。

```
istream &operator>>(istream &stream, 类名 对象名)
{
    //操作代码
    return stream;
}
```

图 14-34 定义运算符“>>”重载函数的一般形式



其参数的说明及使用时需注意的问题与重载输出运算符“<<”一样。

【示例 14-20】下面的程序中重载了输入运算符“>>”和输出运算符“<<”，实现代码及结果如图 14-35 所示。

```
#include <iostream.h>
class point
{
    int x,y;
    public:
        point(int a,int b)
        {
            x=a;
            y=b;
        }
    friend ostream&operator<<(ostream &output, point obj);
    friend istream&operator>>(istream &input, point &obj);
};

ostream &operator<<(ostream &output, point obj)
{
    output<<"x="<<obj.x<<"    "<<"y="<<obj.y<<endl;
    return output;
}

istream &operator>>(istream &input, point &obj)
{
    cout<<"请输入x,y的值: "<<endl;
    input>>obj.x;
    input>>obj.y;
    return input;
}

int main()
{
    point p(10,20);
    cout<<p;
    cin>>p;
    cout<<p;
    return 0;
}
```

定义类

重载构造函数

重载运算符为友元函数

定义友元函数

定义友元函数

创建对象

调用输出运算符

调用输入运算符

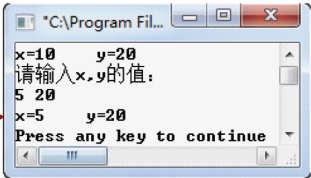


图 14-35 重载了输入运算符“>>”和输出运算符“<<”实例



**注意：**在重载输入运算符时，其对象参数必须是一个引用，即其前面必须加“&”符号，否则输入的值不会传递到类的数据成员中。



## 14.6 命名空间

命名空间是 C++ 新增加的一种功能，其主要作用是防止标识符名称冲突，将逻辑上相关的标识符置于同一命名空间中。在 C++ 中，不仅各种库函数和类使用命名空间来界定，用户也可以定义自己的命名空间。

### 14.6.1 命名空间概述

在 C++ 中，引入命名空间（namespace）概念的原因如图 14-36 所示。

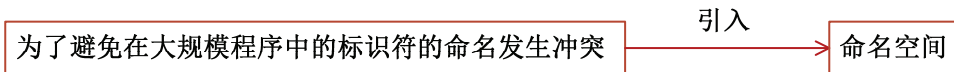


图 14-36 引入命名空间概念的原因

命名空间是为解决 C++ 中标识符的命名冲突而服务的。解决方法如图 14-37 所示。

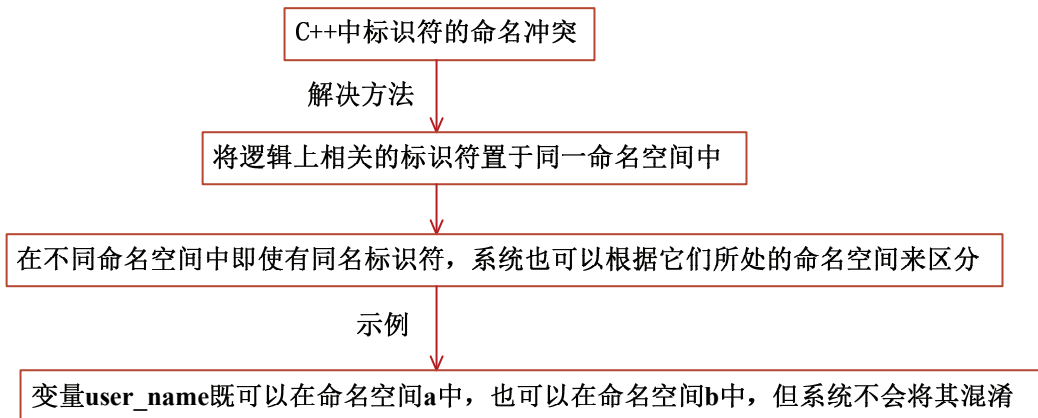


图 14-37 解决 C++ 中标识符的命名冲突

以下是三个与命名空间相关的概念，如图 14-38 所示。

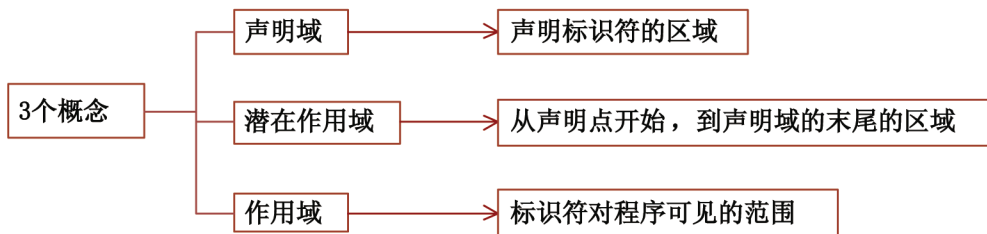


图 14-38 3 个与命名空间相关的概念

### 14.6.2 定义命名空间

在 C++ 中，有两种形式的命名空间：有名命名空间和无名（匿名）命名空间，如图 14-39 所示。

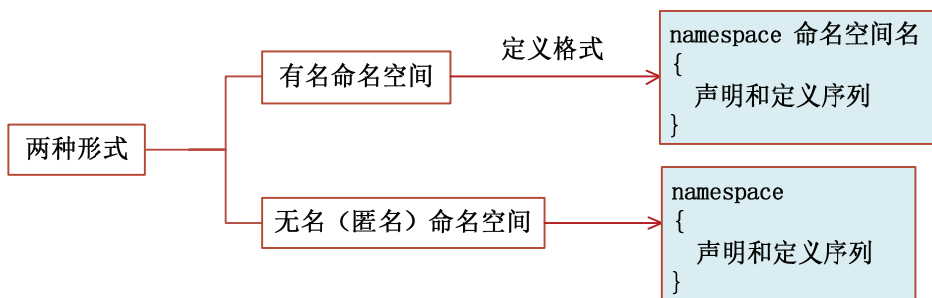


图 14-39 定义命名空间

声明和定义序列内可以声明和定义命名空间的成员，称为内定义。也可以只在命名空间内声明成员，在命名空间外定义成员，称为外部定义。外部定义成员的格式如图 14-40 所示。

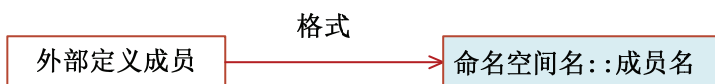


图 14-40 外部定义成员的格式

在定义命名空间时需注意几点，如图 14-41 所示。

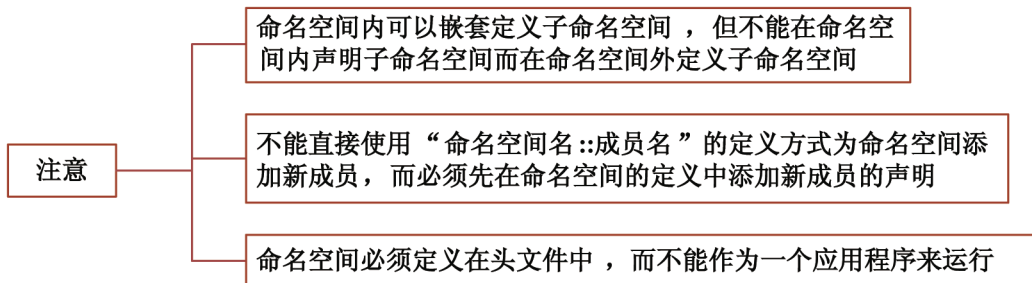


图 14-41 定义命名空间需注意事项

### 14.6.3 使用命名空间

由于命名空间的定义中包含了许多成员的定义，那么在实际的程序设计中，如何来使用命名空间中的这些定义呢？标准 C++给出了 3 种引用命名空间内成员的方法，如图 14-42 所示。

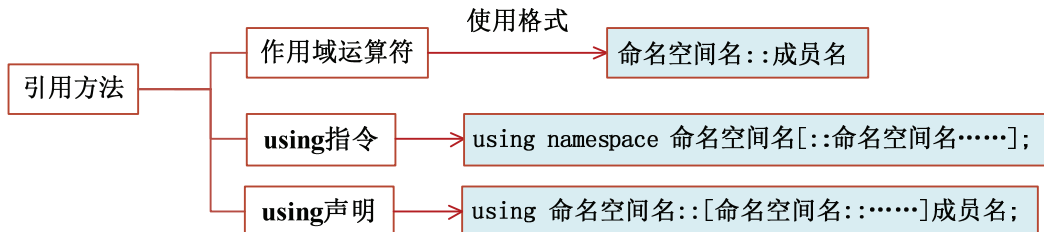


图 14-42 3 种引用命名空间内成员的方法

using 编译指令和 using 声明都可以简化对命名空间中成员的访问。使用 using 指令，可以一劳永逸，对整个命名空间的所有成员都有效。而 using 声明，则必须对命名空间的不同成员逐个去声明。



## 14.7 小结

本章主要介绍了 C++ 输入/输出流的相关内容。首先由 C 语言中的输入/输出函数 `scanf()` 和 `printf()` 的缺陷引出 C++ 中的输入/输出流，接着详细讲解了 C++ 的标准输入/输出流的相关概念和使用，以及 C++ 的输入/输出格式控制。此外，对输入/输出运算符 “>>” 和 “<<” 进行了重载，使其能够进行类对象的输入/输出。最后对命名空间的相关概念和使用进行了讲述。

## 14.8 习题

【题目 14-1】定义两个命名空间，包含一个变量和一个函数，然后在主函数中只使用该名称空间的函数和变量，并输出结果。程序运行结果如图 14-43 所示。

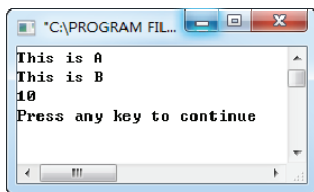


图 14-43 运行结果

【题目分析】本题目要求读者熟悉命名空间的知识。

【关键代码】

```
#include<exception>
#include<iostream>
using namespace std;
namespace A
{
    int n;
    void dispA()
    {
        cout<<"This is A"<<endl;
    }
}

namespace B
{
    void dispB()
    {
        A::dispA();
        cout<<"This is B"<<endl;
    }
}

int main()
{
    A::n=10;
    B::dispB();
    cout<<A::n<<endl;
    return 0;
}
```

【题目 14-2】对输出流进行重载，输出自定义类型日期类，运行结果如图 14-44 所示。

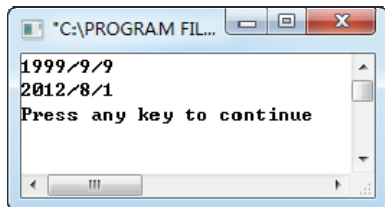


图 14-44 运行结果

【题目分析】考查输出流的重载。

【关键代码】

```
#include <iostream.h>
class CDate
{
private:
    int year;
    int month;
    int day;
public:
    CDate(int y=2008,int m=12,int d=31);
    friend ostream & operator<<(ostream & output,CDate & date);
};

ostream & operator<<(ostream & os,CDate & date)
{
    int a,b,c;
    a=date.year;
    b=date.month;
    c=date.day;
    os<<a<<"/"<<b<<"/"<<c<<endl;
    return os;
}

CDate::CDate(int y,int m,int d)
{
    year=y;
    month=m;
    day=d;
}

int main()
{
    CDate date1(1999,9,9);
    cout<<date1;
    CDate date2(2012,8,1);
    cout<<date2;
    return 0;
}
```

【题目 14-3】对输入流进行重载，输入自定义类型日期类，运行结果如图 14-45 所示。

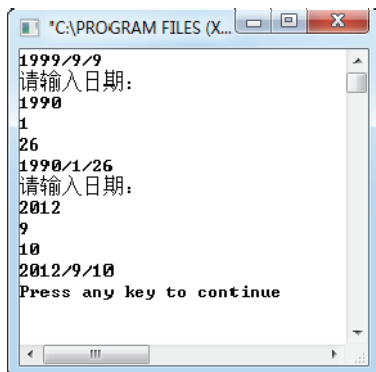


图 14-45 运行结果

【题目分析】考查输入/输出流的重载。

【关键代码】

```
#include <iostream.h>
class CDate
{
private:
    int year;
    int month;
    int day;
public:
    CDate(int y=2008,int m=12,int d=31);
    friend istream & operator>>(istream & input,CDate & date);
    friend ostream & operator<<(ostream & output,CDate & date);
};

istream & operator>>(istream & input,CDate & date)
{
    input>>date.year>>date.month>>date.day;
    return input;
}

ostream & operator<<(ostream & output,CDate & date)
{
    int a,b,c;
    a=date.year;
    b=date.month;
    c=date.day;
    output<<a<<"/"<<b<<"/"<<c<<endl;
    return output;
}

CDate::CDate(int y,int m,int d)
{
    year=y;
    month=m;
    day=d;
}

int main()
{
    CDate date1(1999,9,9);
    cout<<date1;
    cout<<"请输入日期: "<<endl;
    cin>>date1;
```



```
    cout<<date1;
    cout<<"请输入日期: "<<endl;
    cin>>date1;
    cout<<date1;
    return 0;
}
```

**【题目 14-4】** 定义一个复数类，使自定义的复数类型支持输入/输出操作。

**【题目分析】** 考查输入/输出流的重载。

**【关键代码】**

```
class complex
{
private:
    double real;
    double imag;
public:
    friend istream & operator>>(istream & input,complex &);
    friend ostream & operator<<(ostream & output,complex &);
}
istream & operator>>(istream & input,complex & c1)
{
    input>>c1.real;
    while(input.get()!='*')
    {
    }
    cin>>c1.imag;
    return input;
}
ostream & operator<<(ostream & output,complex & c1)
{
    output<<c1.real<<" + i*"<<c1.imag<<endl;
    return output;
}
```



# 第 15 章 预处理和宏

程序在编译之前先要经过预编译阶段。预编译的任务是根据程序中的宏指令补充和完善源代码。有了宏指令，可以使得某些源代码编辑任务变得轻松，同时也可以控制哪些源代码需要编译，哪些不需要编译。

## 15.1 预处理概述

预处理器是专门用来处理宏指令的程序。在编译器运行之前，会先运行预处理器，查找所有的预处理指令。预处理指令以“#”开头，而且不以“;”结束，以区别于一般的语句。预处理器根据预处理指令生成新的源代码文件（临时文件，可以通过编译器的选项输出到指定目录中）。

编译器的作用是把源代码转换成汇编语言或机器指令。但是，编译器并不是直接编译程序员写成的源文件，而是编译经预处理器处理后所产生的新的源文件。这些新的源文件经过编译器生成目标文件，再经过链接器生成最终的可执行程序。

预处理器的任务就是执行源代码中的预处理指令，并对源代码进行相应的处理。因此，从预处理指令的类型来讲，预处理器的任务有这几部分：将其他文件包含到当前文件中、定义宏、定义类似函数的宏、实施条件编译。接下来，将详细介绍预处理器的各项任务。

## 15.2 宏

所谓宏，是程序中定义的用于替换复杂文本的简短文本。宏定义的一般格式如图 15-1 所示。

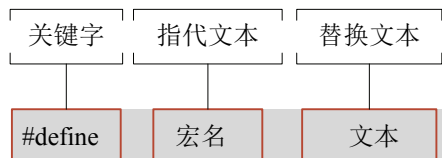


图 15-1 宏定义的一般格式

宏名，即宏的名称，在预编译时被可替换文本替换。可替换文本，即宏名所指代的文本内容。在#define、宏名和可替换文本之间用空格（制表符）分隔。预编译程序将#define 之后的第一个和第二个空格之间的文本作为宏名，其后所有文本作为“可替换文本”，而不管中间有多少个空格。



## 15.2.1 宏展开

在程序的预编译期，预编译程序会解析源代码文本，执行替换源程序的动作，把宏引用的地方替换成定义处的文本。这个动作称为宏的展开。如图 15-2 所示的这段程序是宏的一个简单例子。

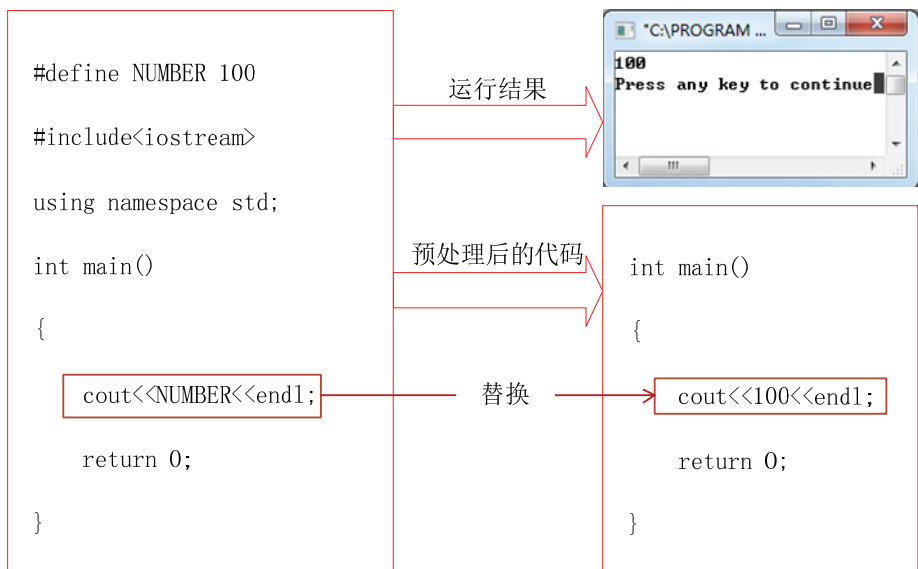


图 15-2 宏的使用

cout 后面的 NUMBER 在预编译时会被替换成 100。



**注意：**在源代码文本中，并不是所有的宏名出现的地方都会进行展开。如果这个宏名出现在一个双引号中，则该宏名就成了字符串的一部分，也就不会进行展开了。

## 15.2.2 替代常量

用宏替代字面常量，其好处是直观、简洁、修改方便。譬如对于圆周率，其值是一个无理常量 3.1415926…。如果在程序中每一处要使用圆周率的地方都直接书写这个常量，那么源代码修改起来就不太方便。一旦要求改变数字的精度，则所有使用的地方都要修改。无疑修改量就比较大，而且也不能保证每一处都修改对。

如果使用宏替换字面常量，而程序中使用圆周率的地方都使用宏而非圆周率本身，那么当修改圆周率数值时，只要修改这个宏定义即可，大大减少了编程人员的工作量。如图 15-3 所示的代码，用宏替换常量 3.1415926。

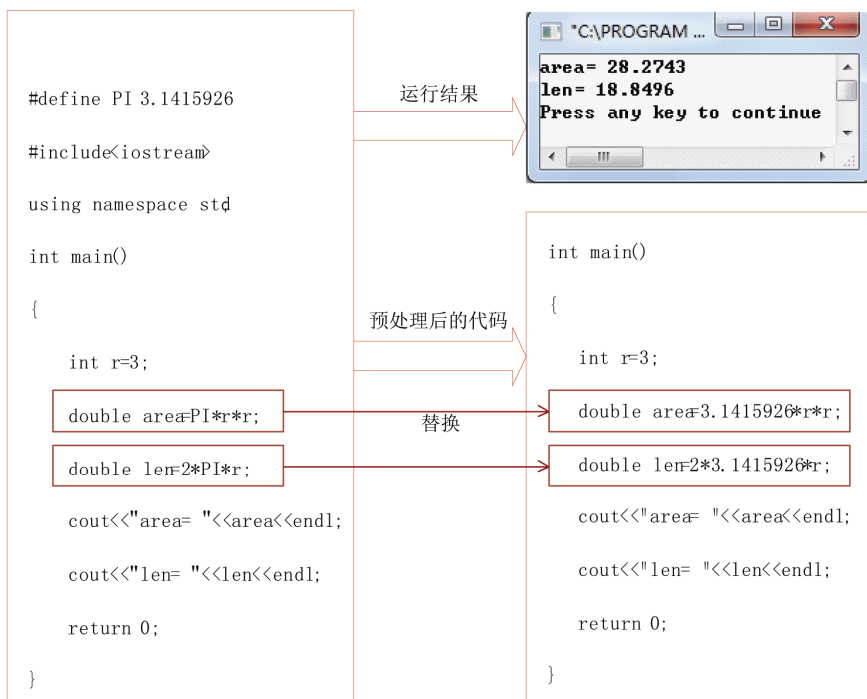


图 15-3 宏替换字面常量

图 15-3 中使用宏 `PI` 替换了常量 `3.1415926`，若程序中的常量需要变化，则只需要更改宏定义即可。

相对于常量，使用宏也有缺点。字面常量在编译时处理，有类型信息。而宏则是在预编译时展开的，只是进行单纯的文本替换，没有类型信息。因此对于一些类型要求比较严格的地方，使用宏有一定的风险。

定义宏时可以使用前面已经定义的宏。例如，假设已经存在一个宏 `NUMBER`，其替换的文本是 `100`，当定义新的宏时，可以在宏的可替换文本中使用 `NUMBER`，如图 15-4 所示。

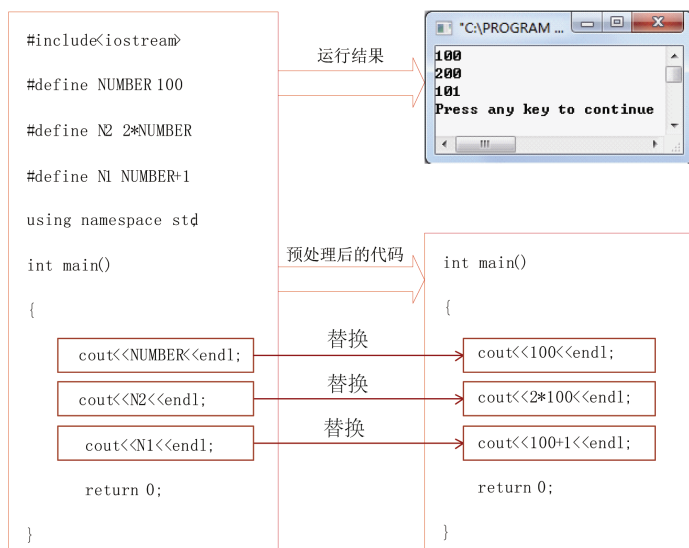


图 15-4 宏展开



### 15.2.3 替代运算符

除了可以用宏替代字面常量，还可以用宏替代某些运算符，包括加减乘除、逻辑与和逻辑非等，甚至函数和语句块的花括号。利用这些宏定义，可以编写出貌似违反 C++ 语法、但实际上合法的源代码。例如，如图 15-5 所示的代码。

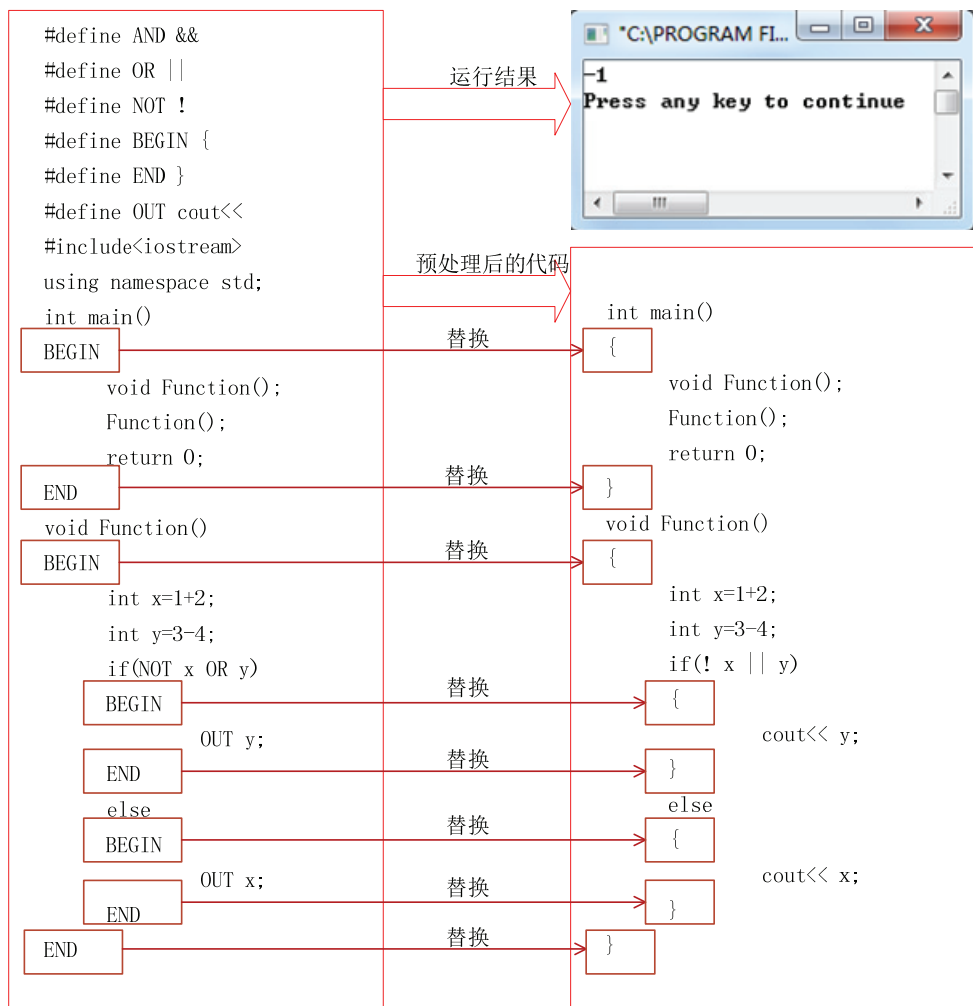


图 15-5 宏替代运算符



## 15.3 带参数的宏

简单的宏定义只能进行简单的文字替换，扩展能力有限。如果宏能够像函数那样带有参数，并根据参数的不同自动展开成不同的文本，则宏的扩展能力将大大提高。实际上，在 C++ 标准中，可以利用预处理命令 `#define` 定义带参数的宏。



### 15.3.1 定义带参数的宏

有参数宏的宏名后需要带参数，其语法格式如图 15-6 所示。



图 15-6 带参数宏的定义 1

若可替换文本一行写不完，可以分成多行，并在每一行的末尾加上分行的符号“\”（除了最后一行）。其语法格式如图 15-7 所示。

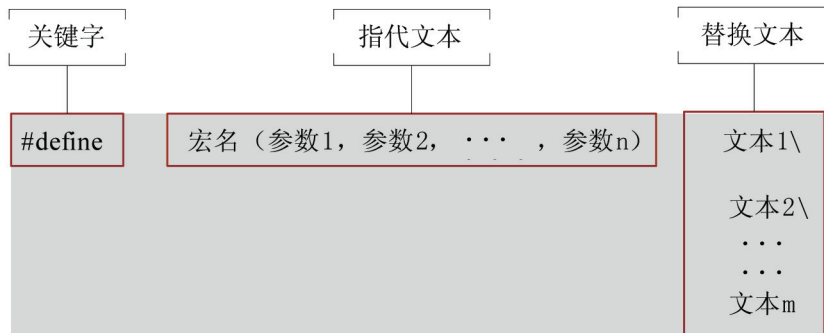


图 15-7 带参数宏的定义 2



**注意：**要定义带参数的宏，在宏名和左括号之间不能有空格（或制表符），否则就成了普通的宏定义，括号及其里面的内容也将会成为可替换文本的一部分。但是，在括号中的各个参数之间可以任意添加空格。

在可替换文本中可以引用括号中的参数，从而根据参数的不同，展开成不同的源代码文本。例如，下例定义了一个宏，接收两个参数，然后比较两个参数的大小，并输出其中的较小值，示例代码如图 15-8 所示。

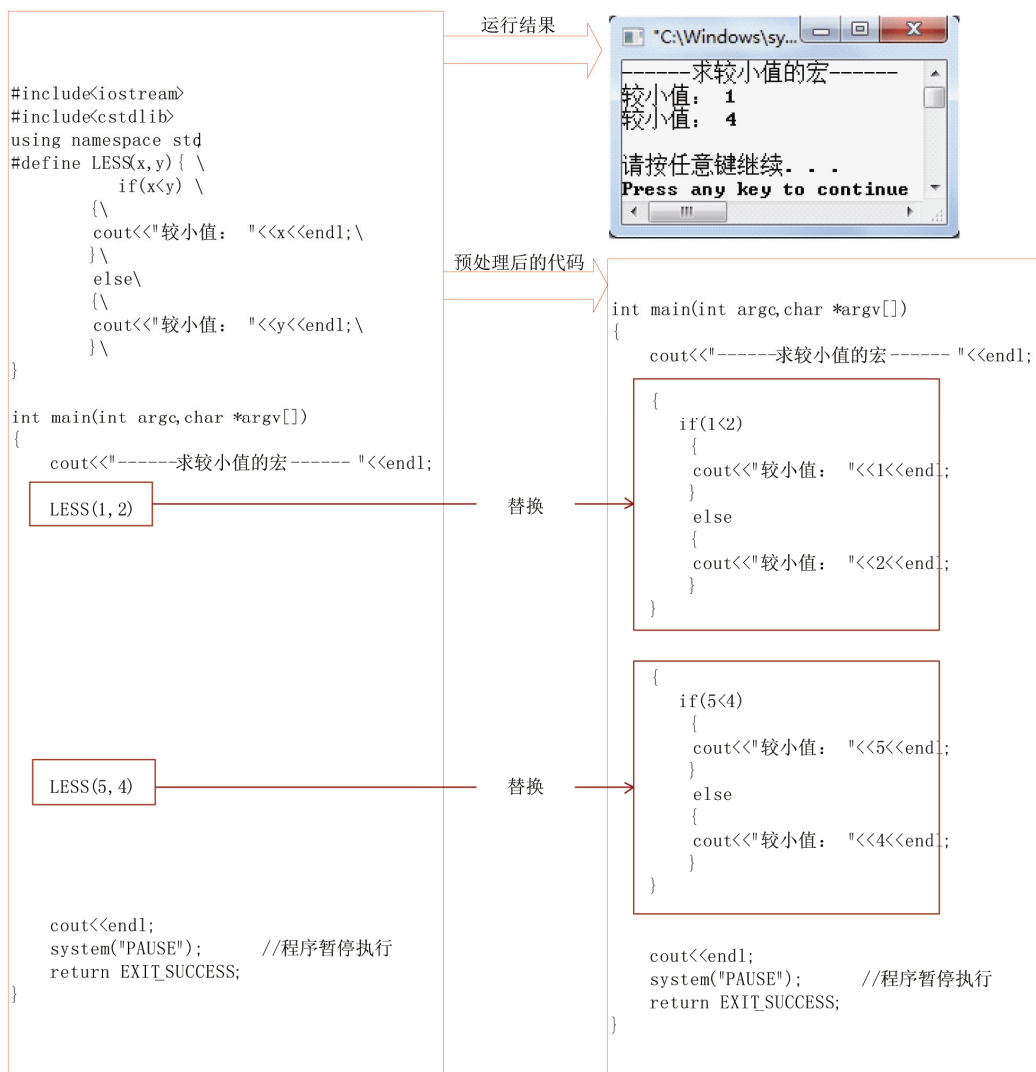


图 15-8 带参数的宏



**注意：**在上述例子中使用了带参数的宏 `LESS`，但是在语句的结尾处并没有分号。乍一看好像不符合 C++ 的语法规范，但是实际上并没有错。这是因为宏是在预编译时处理的，而不是编译期。而且，宏展开后的结果符合 C++ 语法规范，所以这些语句的结尾处没有分号并不会出错。

### 15.3.2 注意宏展开的结果

宏的行为有时和所认知的行为会有些不同，主要原因是通常认为宏作为语句块，会优先执行，但这是不对的，宏并没有那么“聪明”，或者说编译器并没有那么“聪明”，预编译器



所做的只是忠实地将宏展开。我们来看图 15-9 所示的程序，预编译器不会自作聪明地为“x+x”加括号，因此最后输出的结果是 30。

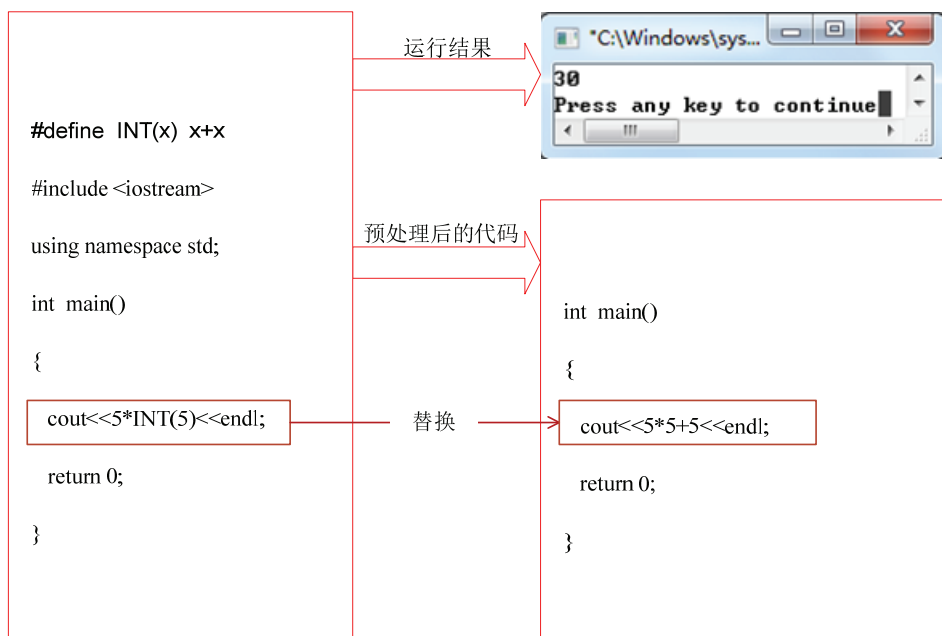


图 15-9 带参数宏的展开

如果凭第一印象，很可能会做这样的计算： $5 * (5 + 5) = 50$ 。但是，预编译器并不这么认为，而会做这样的解释： $5 * 5 + 5 = 30$ ，忠实地将原来的宏展开。所以读者写宏时一定要小心，预防宏的行为不符合预期。

若想让结果输出为 50，只要给宏加上括号即可，如图 15-10 所示。

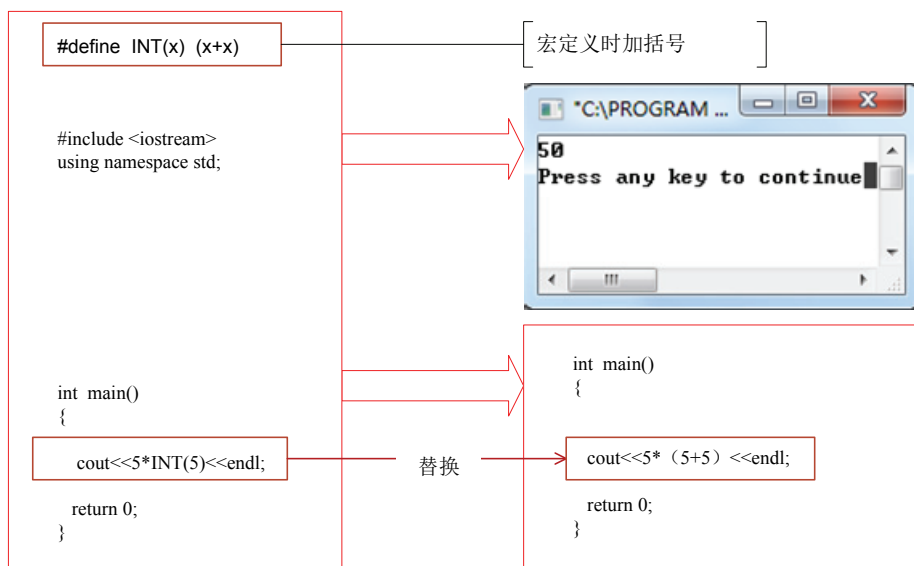


图 15-10 展开宏时注意括号



图 15-10 中的 “cout<<5\*INT(5)<<endl;” 经预处理后，转换为 “cout<<5\*(5+5)<<endl;”。

15.3.3 带参数的宏与函数的比较

带参数的宏在定义和使用方面同函数非常相似，都可以接收参数，并且可以根据不同的参数产生不同的结果。但是这两者毕竟是不同的东西，存在很大差异，例如：

- (1) 函数在运行时依然存在，但宏只存在于预编译期间，程序运行时并不存在。
- (2) 函数所占内存空间只有一份，但如果宏被调用多次，则其语句也将重复多次。
- (3) 函数调用有时间和空间方面的开销，但宏没有。
- (4) 函数接受参数的传递，但宏只是对参数的简单替换。
- (5) 函数的参数有类型信息，但宏的参数没有类型信息。
- (6) 函数可以调试，宏不可以。

15.4 条件编译

条件编译指令可以对程序源代码的各部分有选择地进行编译，该过程就称为条件编译。常见的条件编译指令有 #if、#elif、#else、#ifndef、#ifdef、#endif 等。它们主要用来在编译时进行选择，屏蔽掉一些代码。下面将对条件编译内容进行相关介绍。

15.4.1 宏指令

C++中的宏指令都是在 ANSI 标准中的，表 15-1 列出了常见的宏指令。

表 15-1 常见的宏指令

#define	#error	#include	#if
#else	#elif	#endif	#ifdef
#ifndef	#undef	#line	#pragma

#define 在前面已经介绍过了，这里就不再讨论。#error 可以强迫编译程序停止编译，用来在编译期间检查环境是否符合要求或者与约束的条件发生了冲突，其使用格式如图 15-11 所示。

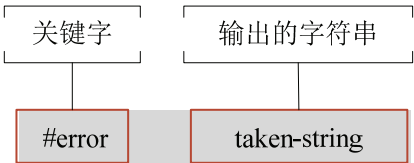


图 15-11 强迫停止编译指令#error

当程序遇到这个关键字时，就会停止编译，产生一个错误信息，并且输出后面的 token-string。例如：

```
#if !defined(__cplusplus)
#error C++ compiler required.
#endif
```





上面这段代码的意思是在预编译期间检查当前的程序是否为 C++ 编译环境，如果不是，就定义 `#error`，让编译器停止编译。`#if` 和 `#endif` 将在后面介绍，这里只需要将 `#if` 视为普通的 `if` 判断语句，将 `#endif` 看成 `if` 判断的结束。

`#include` 使编译程序将 `#include` 所指向的源文件导入进当前的源文件，被包含的文件必须被尖括号或者引号包围起来。`#if`、`#else`、`#elif`、`#endif`、`#ifdef` 和 `#ifndef` 属于条件编译命令，可以对程序的各个部分有选择地进行编译。`#undef` 命令用来取消前面定义过的宏名。下面看一个宏使用的例子，如图 15-12 所示。

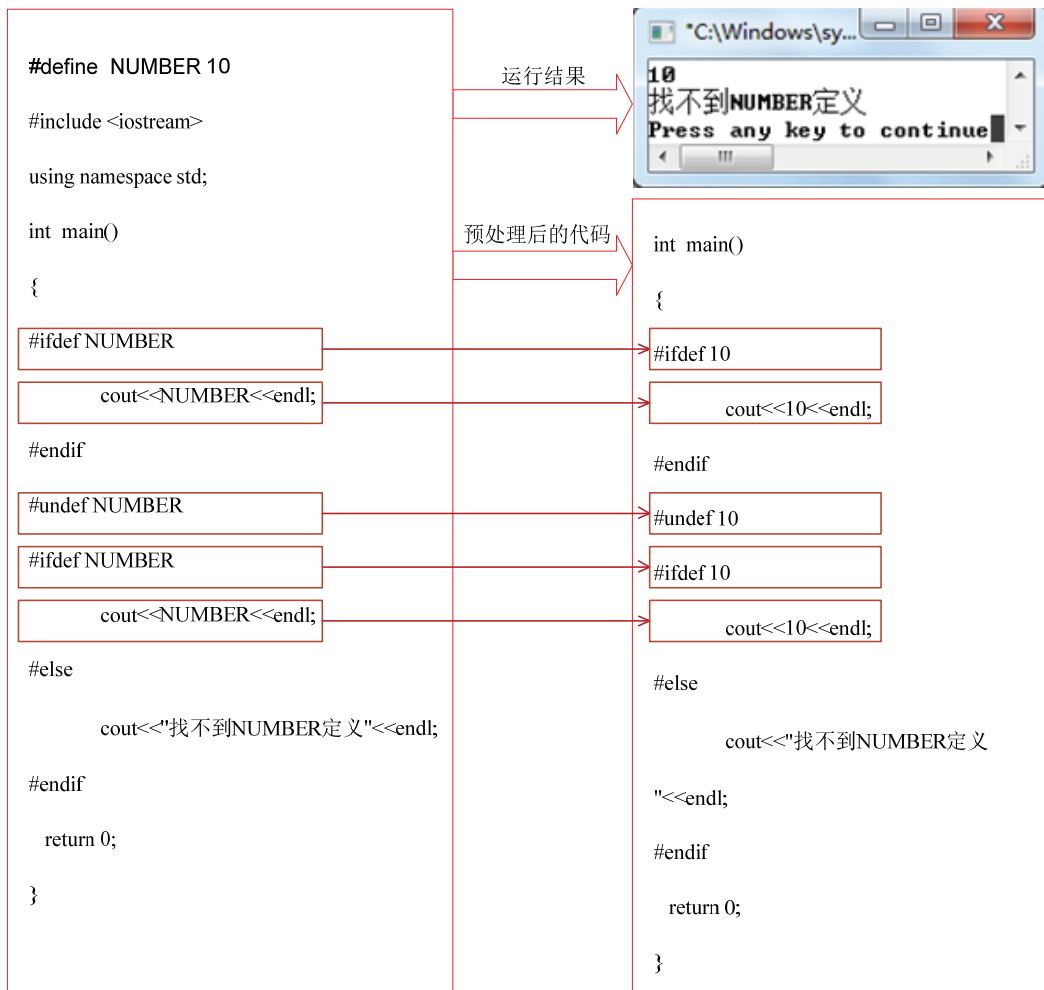


图 15-12 宏指令的使用

在程序的第 1 行定义了 `NUMBER` 宏，在第 6 行输出 10。在第 8 行取消了 `NUMBER` 宏的定义，所以后面会输出“找不到 `NUMBER` 的定义”。



## 15.4.2 使用条件编译

#if、#else、#elif、#endif、#ifdef 和 #ifndef 都是条件编译命令。所谓条件编译，就是可以将源文件中的代码分成几个部分，有选择地编译各个部分。对于前 3 个宏，可以理解为 if、else 和 else if，#endif 表示这个条件编译选择的结束。看图 15-13 中的代码。

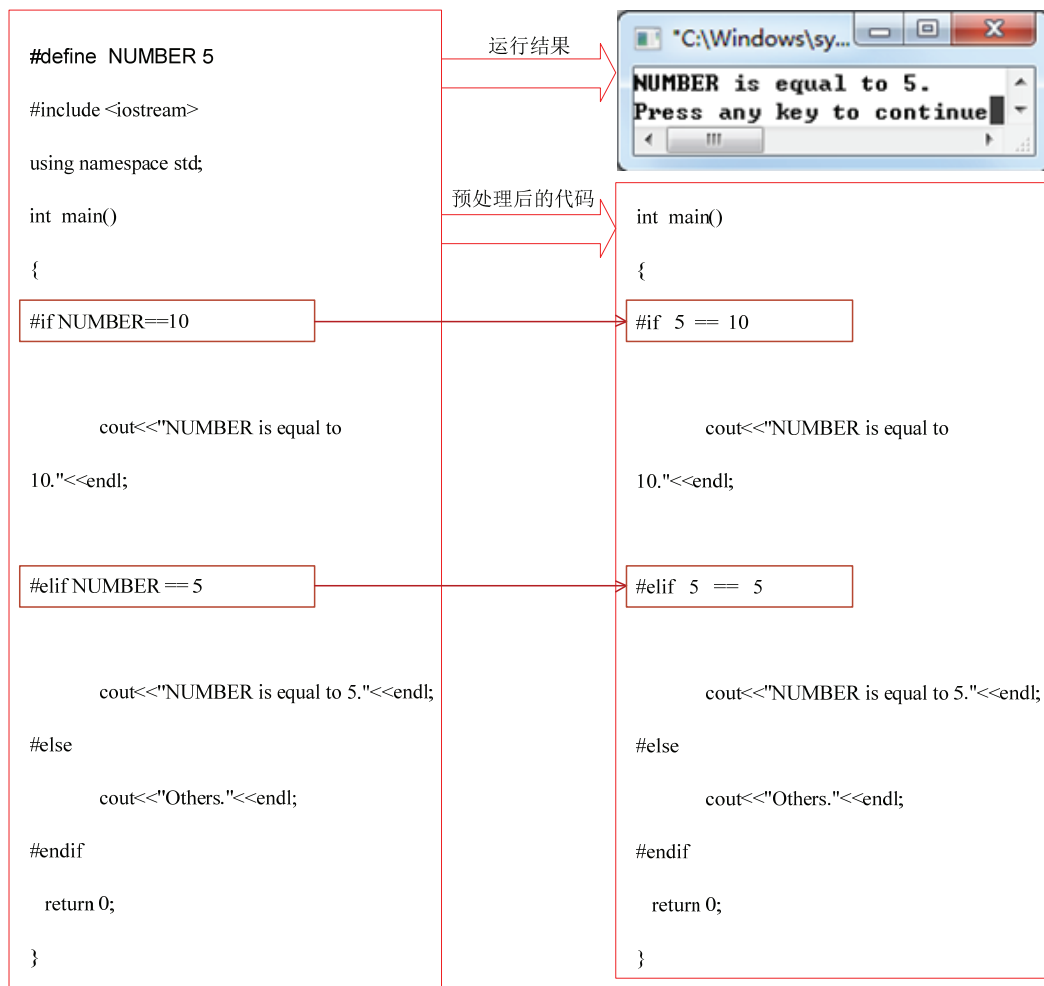


图 15-13 用 #if、#else 和 #elif 进行条件编译

程序的第一行定义 NUMBER 宏，令其等于 5。后面的代码判断是否等于相应的值，选择性地编译后面的语句，由于定义 NUMBER 等于 5，编译器输出“Number is equal to 5”。

另一种条件编译的方式就是使用 #ifdef 和 #ifndef。同样，这两个命令也用 #endif 作为作用域的结束。#ifdef 判断后面的标识符是否被定义，通常都是指预定义的宏，#ifndef 就是 #ifdef 的取反。下面看如图 15-14 所示的程序。



```
#define DEBUG

#include <iostream>

using namespace std;

int main()
{
    #ifdef DEBUG    //判断DEBUG是否被定义

        cout<<"DEBUG version."<<endl;

    #endif

    return 0;
}
```

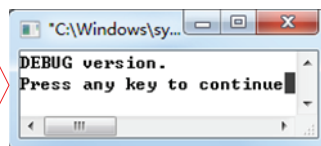


图 15-14 用#ifdef 和#endif 进行条件编译

第 4 行判断是否定义 DEBUG 标识符（在第一行定义），然后编译后面的语句。再来看一个例子，示例代码如图 15-15 所示。

```
#include<iostream>

#define WINDOWS

using namespace std

void disp()
{
    #if defined(WINDOWS)

        cout<<"本程序当前运行在 Windows环境下。"<<endl;

    #else

        cout<<"本程序当前运行在非 Windows环境下。"<<endl;

    #endif

}

int main()
{
    disp();

    return 0;
}
```

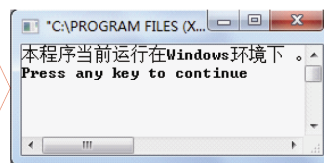


图 15-15 条件编译 1



代码第 4 行的 `disp` 函数采用条件编译指令编写，因为程序中定义了 `WINDOWS` 宏，所以输出“本程序当前运行在 Windows 环境下”，假如将“`define WINDOWS`”语句注释掉，程序将输出“本程序当前运行在非 Windows 环境下”，如图 15-16 所示。

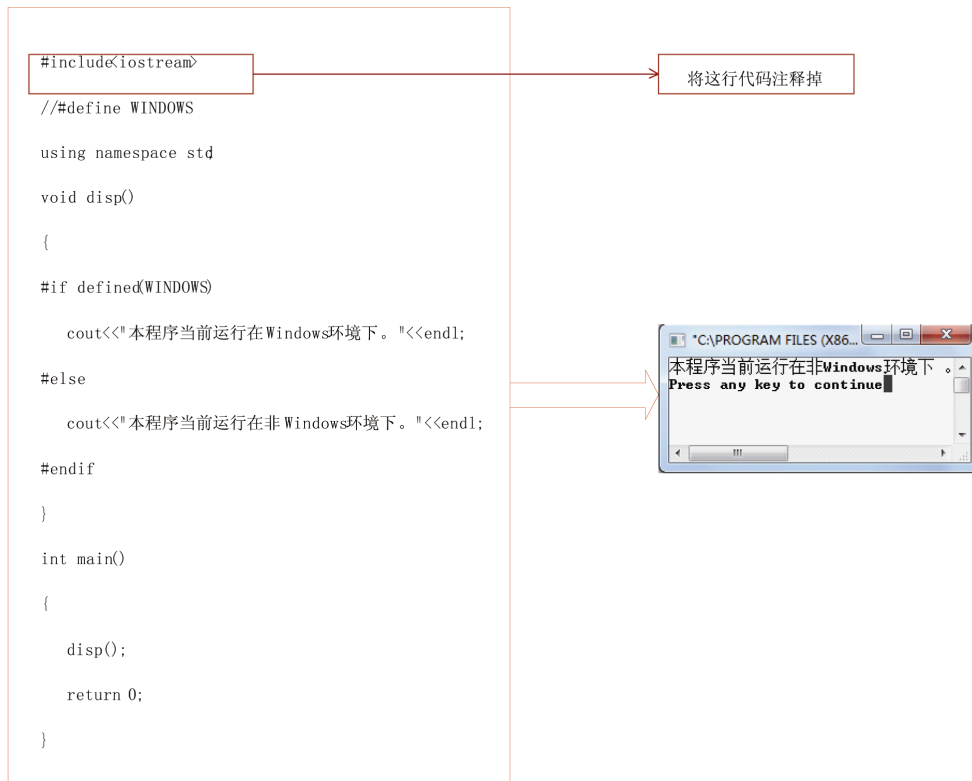


图 15-16 条件编译 2

代码第 6 行“`#if defined(WINDOWS)`”语句中的“`define`（宏名）”的含义是若宏被定义则返回 1，否则返回 0，`defined` 之间可以进行逻辑运算。

## 15.5 文件包含和头文件卫士

### 15.5.1 包含文件指令

包含文件的预处理指令是“`#include`”，其作用是将别的文件包含到当前文件中。在实际使用中，一般被包含的文件是头文件。头文件中一般是函数、类等声明，包含到当前文件中后，就可以在当前文件中引用头文件中的函数、类等。例如，源代码文件进行预编译后产生的临时源代码文件如图 15-17 所示。

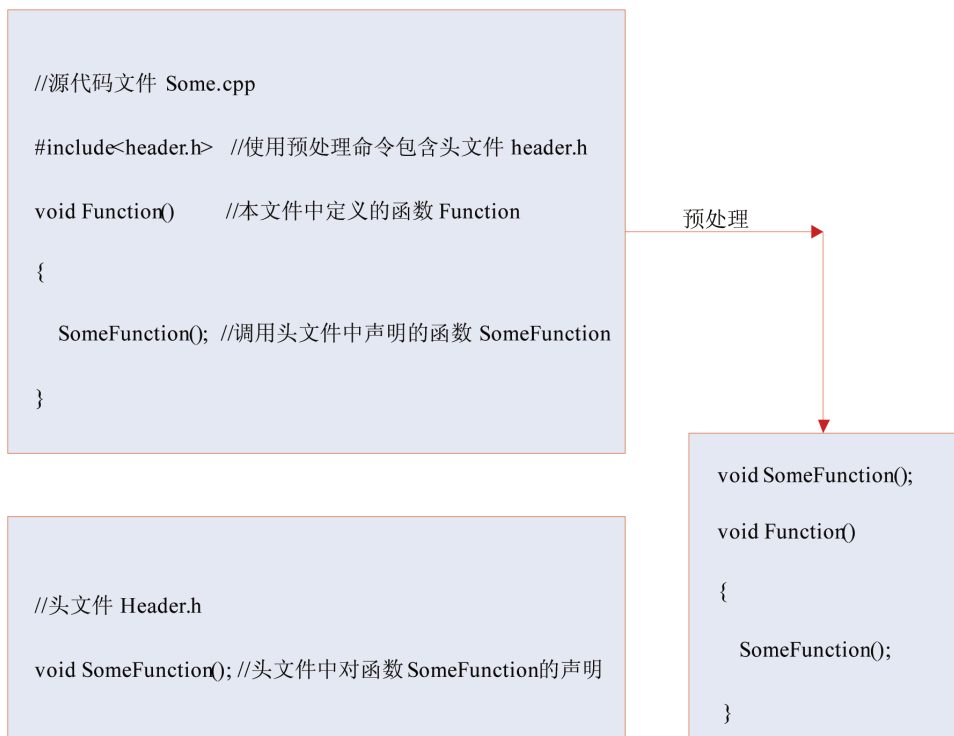


图 15-17 包含头文件

按照 C/C++ 的语法要求，要使用某个标识符（变量、函数、类等），必须在使用之前先声明。虽然在 `Some.cpp` 文件中没有函数 `SomeFunction` 的声明，但是由于包含了头文件 `header.h`，所以仍然可以使用该函数。

### 15.5.2 搜索头文件

使用“`#include`”指令包含头文件时，其后的头文件有两种方式，一种是使用双引号，另一种是使用尖括号，如图 15-18 所示。

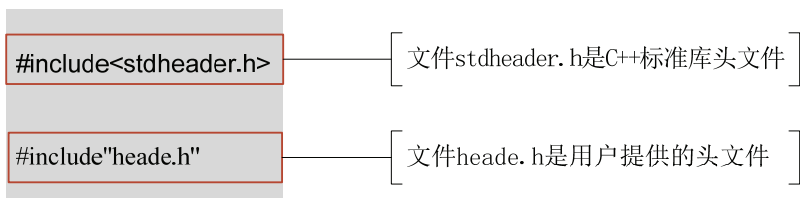


图 15-18 头文件的两种包含方式

如果文件名用尖括号括起来，表明这个文件是一个工程或 C++ 标准库头文件。预编译器会首先搜索在工程中预定义的目录，然后搜索 C++ 编译器的安装目录。可以通过设置工程搜索路径环境变量或命令行选项来修改。

如果文件名用一对引号括起来，则表明该文件是用户提供的头文件。预编译器首先从当前文件目录开始搜索，如果找不到，就从工程中定义的目录和编辑器的安装目录中查找。



### 15.5.3 头文件卫士

当工程中文件较多时，很可能出现一个头文件被多次包含的情况。但是，C++中同一个类型被声明两次是非法的，来看一个例子。如图 15-19 所示，文件 `Animal.h` 中定义了类 `CAnimal`，文件 `pig.h` 中定义了类 `CPig`，文件 `horse.h` 中定义了类 `CHorse`，主函数中用类 `CPig` 和 `CHorse` 分别定义了变量 `pig` 和 `horse`。

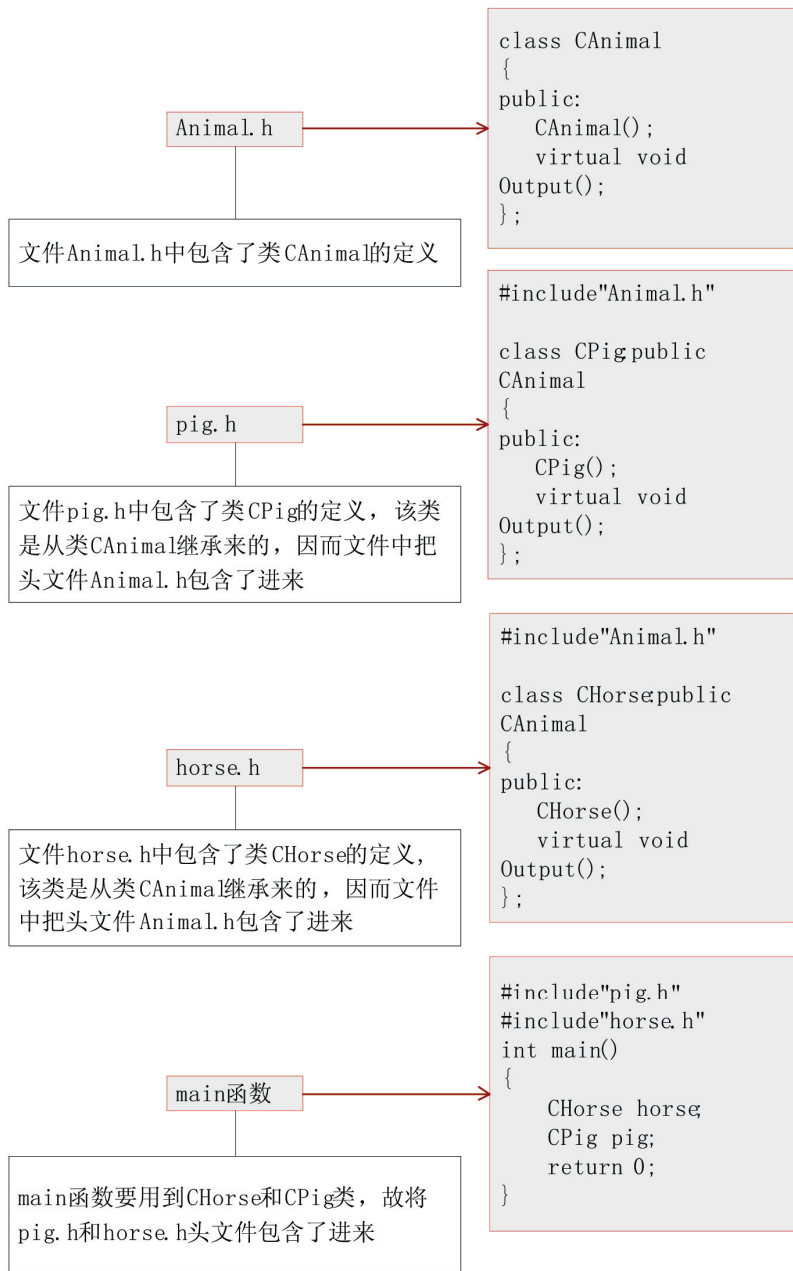


图 15-19 重复定义 1



在 main 函数中使用了 CHorse 和 CPig 两个类，所以需要包含这两个类的头文件，但是，这两个类都包含 CAnimal 的定义，编译器不知道该选择哪个，会报告重复定义的错误。如图 15-20 所示，对上述代码进行编译，编译器给出了错误信息。

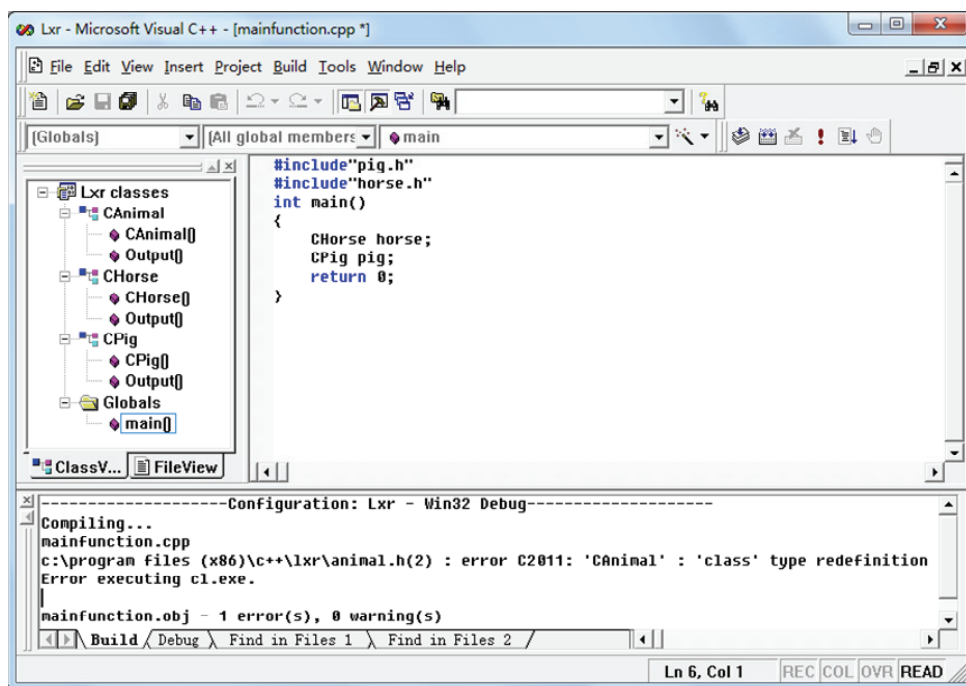


图 15-20 重复定义 2

为了避免因重复包含头文件而重复定义类型，导致编译失败，这时就需要头文件卫士的帮助。所谓头文件卫士，就是用一组宏命令将头文件包含起来，使其不会被重复包含，看 Animal.h 的例子：

```
#ifndef ANIMAL_H
#define ANIMAL_H

#endif
```

#ifndef 是定义在头文件所有内容之前的，#endif 是定义在所有内容之后的，用预编译命令 #ifndef 和 #endif 将整个 Animal.h 头文件中的内容包含起来，这样便不会有编译错误了，下面来分析一下。

当 main 函数所在文件第一次包含 pig.h 文件时，同时会导入 Animal.h 中的内容，这时预编译器分析当前文件没有定义 ANIMAL\_H，就会在当前文件中定义。当再次包含 horse.h 文件时，导入 Animal.h，发现文件中已经定义了 ANIMAL\_H，就会查找 #else 或者 #endif，这时会直接跳转到 #endif，不会包含当前文件的任何内容。

同理，如果想继续从 Horse 上继承，例如 Whitehorse 或 Blackhorse 之类的，在一个地方同时使用，这时就需要在 horse.h 中加上头文件卫士。通常的习惯是在所有的头文件中都加入头文件卫士。



## 15.6 预定义的宏

在 C++ 标准中，规定了一些预定义的宏。即这些宏不需要由开发者定义，而是由预编译器提供，开发者只要使用即可。表 15-2 列出了一些常用的预定义的宏。

表 15-2 常用的预定义的宏

预定义的宏	含义
__FILE__	代表当前源代码文件名的字符串文字
__LINE__	代表当前源代码中的行号的整数常量
__DATE__	进行预编译的日期（"Mmm dd yyyy"形式的字符串文字）
__TIME__	源文件预编译时间，格式"hh: mm: ss"
__func__	当前所在函数名

上述预定义的宏两侧都是两个下画线，而不是一个。

其中，宏 \_\_LINE\_\_ 表示该宏所在行的行号。这个行号可以用预处理命令 #line 进行修改。下例使用预定义的宏输出某些源代码的信息，如图 15-21 所示。

```
#include <cstdlib>

#include <iostream>

using namespace std           //使用名称空间

std

int main(int argc,char *argv[]) //主函数
{

    cout<<"--输出编译信息--"<<endl;

    cout<<" 文件名:      " <<__FILE__<<endl;

    cout<<" 行号:      " <<__LINE__<<endl;

    cout<<" 预编译日期:  " <<__DATE__<<endl;

    cout<<" 预编译时间:  " <<__TIME__<<endl;

    #line 100 "测试文件"

    cout<<"--修改文件名和行号 --"<<endl;

    cout<<" 文件名:      " <<__FILE__<<endl;

    cout<<" 行号:      " <<__LINE__<<endl;

    system("PAUSE");           //暂停程序

    return EXIT_SUCCESS;

}
```

图 15-21 预定义的宏的使用





程序通过使用预定义的宏，将源文件的文件名、行号、预编译日期和预编译时间进行了输出，然后又修改了文件名和行号。

## 15.7 小结

本章讲解了 C++ 的预处理指令及其用法。预处理是指在编译前所做的处理工作，这类指令主要有文件包含、宏替换、条件编译、布局控制四大类，但布局控制比较复杂，这里不做介绍。综合运用这些指令可以帮助程序员设计出可移植性好、功能强大，适应多种环境的优秀程序。

## 15.8 习题

【题目 15-1】有一个圆球，现在要计算圆球的表面积和体积。小明编写了以下程序求圆球的表面积和体积。

```
#include <iostream.h>
int main()
{
    double r;
    cout<<"请输入圆球的半径: "<<endl;
    cin>>r;
    double s;
    s=3.0*4.0*r*r;
    cout<<"圆球的表面积为: "<<endl;
    cout<<s<<endl;
    double v;
    v=4.0*3.0*r*r*r/3.0;
    cout<<"圆球的体积为: "<<endl;
    cout<<v<<endl;
    return 0;
}
```

用圆周率分别为 3.0、3.1、3.14、3.145、3.1415926 进行计算。若用上面的程序代码，则改起来会很麻烦，而且容易改错。请你帮小明出主意。

【题目分析】本题主要考查宏替换常量的知识。

【关键代码】

```
#define PI 3.0
#define PI 3.1
#define PI 3.14
#define PI 3.145
#define PI 3.1415926
```

【题目 15-2】对于下面的程序，在编译、链接、运行后，输出结果如图 15-22 所示，想想为什么会这么输出。

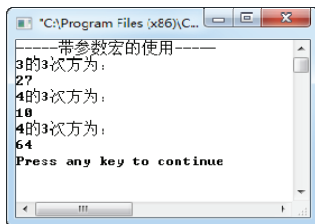


图 15-22 运行结果



```
#include <iostream.h>
#define CUBE(x) x*x*x
int main()
{
    int res=0;
    res=CUBE(3);
    cout<<"3 的 3 次方为: "<<endl;
    cout<<res<<endl;
    res=CUBE(3+1);
    cout<<"4 的 3 次方为: "<<endl;
    cout<<res<<endl;
    res=CUBE(4);
    cout<<"4 的 3 次方为: "<<endl;
    cout<<res<<endl;
    return 0;
}
```

**【题目分析】** 本题主要考查带参数宏的使用问题。

**【关键代码】**

```
#define CUBE(x) x*x*x
res=CUBE(3);
res=CUBE(3+1);
res=CUBE(4);
```

# 第 16 章 标准模板库

STL (Standard Template Library)，即标准模板库，是一个具有工业强度的、高效的 C++ 程序库。该库包含了诸多在计算机科学领域中常用的基本数据结构和基本算法，为广大 C++ 程序员提供了一个可扩展的应用框架，高度体现了软件的可复用性。本章将详细介绍 STL 的相关知识。

## 16.1 标准模板库概述

标准模板库 (STL) 是最新的 C++ 标准函数库中的一个子集，这个庞大的子集占据了整个库大约 80% 的分量，本节将简要介绍标准模板库。

### 16.1.1 C++ 标准库

一般来说，C++ 标准库可分为两部分，如图 16-1 所示。

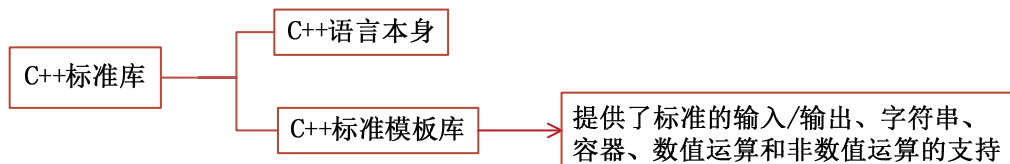


图 16-1 C++ 标准库

C++ 标准库为 C++ 程序员提供了一个可扩展的基础性框架，用户可以通过继承 C++ 标准库获得极大的便利。C++ 标准库主要由几个组件构成，如图 16-2 所示。

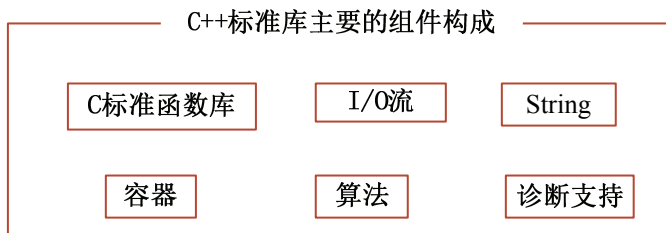


图 16-2 C++ 标准库主要的组件构成

在 C++ 标准库中，实现容器和算法的部分就是标准模板库 (STL)，其中，迭代器让容器和算法这两部分共同工作。整个 C++ 标准库的组成如图 16-3 所示。

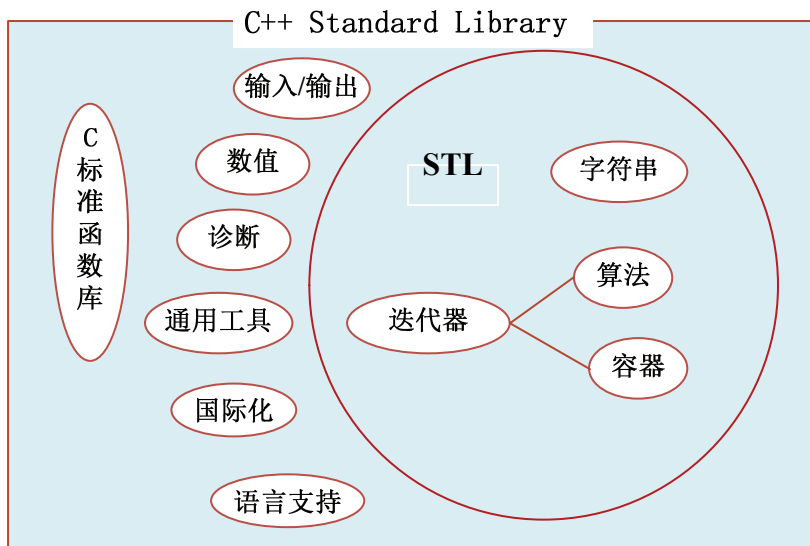


图 16-3 整个 C++ 标准库的组成

### 16.1.2 STL 的形成

STL 是以模板为基础的一套标准库，是 C++ 标准库的一个组成部分。可以将其看做一套支持泛型编程的、兼顾效率和易用性、设计精巧的工具集。其形成经过了二三十年的发展，并拥有各种实现版本，STL 的发展历史如图 16-4 所示。

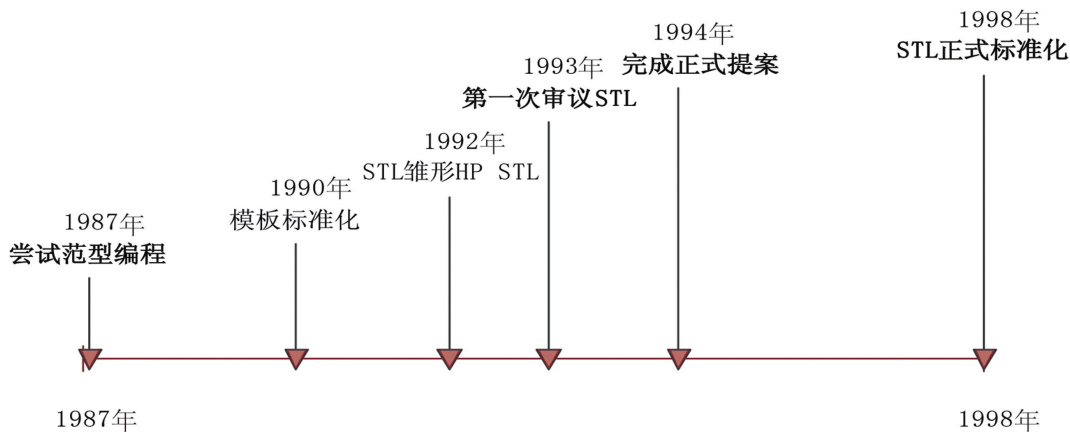


图 16-4 STL 的发展历史

### 16.1.3 STL 的组成

STL 是以模板形式提供的编程组件，解决了很多基础性的编程问题，如数据的组织、查找、计算等。通过使用 STL，可以让开发人员将主要精力集中在程序的高层逻辑上，而不是底层的操作，这样可以较大地提高开发效率。一般来说，STL 由 6 部分组成，如图 16-5 所示。

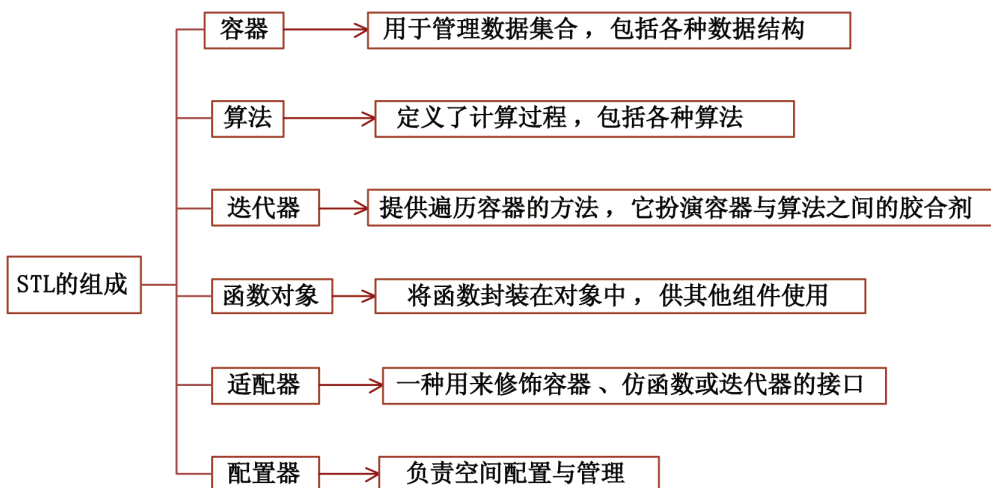


图 16-5 STL 的组成

各个部分之间的关系如图 16-6 所示。

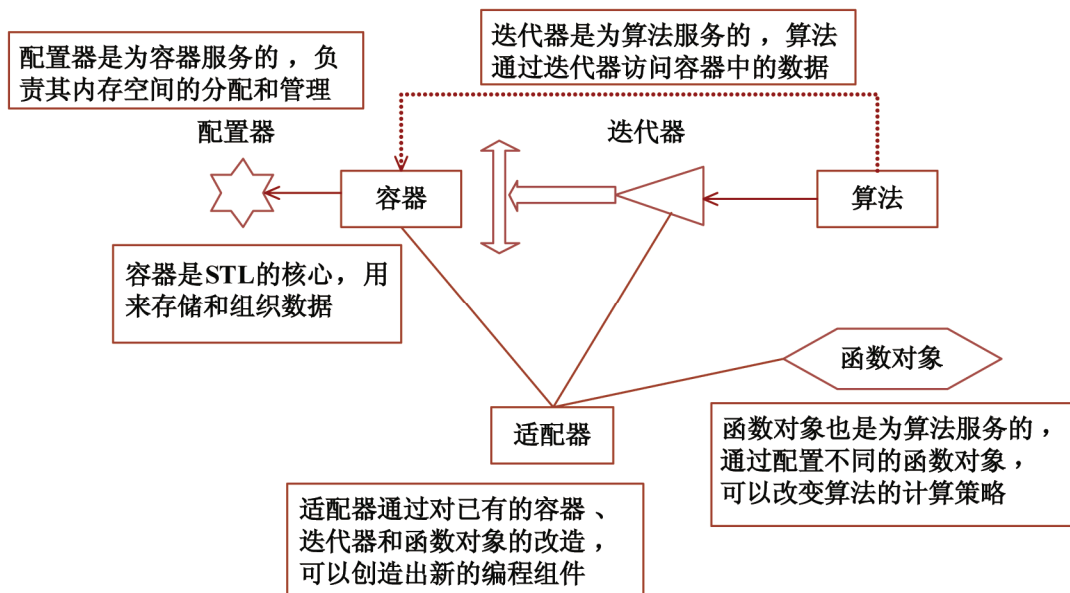


图 16-6 STL 各组成部分之间的关系

在 C++ 标准中，STL 被组织为 13 个头文件：<algorithm>、<deque>、<functional>、<iterator>、<vector>、<list>、<map>、<memory>、<numeric>、<queue>、<set>、<stack>和<utility>。

### 16.1.4 STL 的引入

前面讲到了 STL 的组成，现在通过两个示例来看一下 STL 在实际程序中所起的作用。

【示例 16-1】下面的程序中调用了 STL 中的函数，其实现代码及结果如图 16-7 所示。



```
#include <vector>
#include <iostream>
int main()
{
    std::vector<double> a;

    std::vector<double>::const_iterator i;

    a.push_back(1);
    a.push_back(2);
    a.push_back(3);
    a.push_back(4);
    a.push_back(5);

    for(i=a.begin(); i!=a.end(); ++i)
    {
        std::cout<<(*i)<<std::endl;
    }
    return 0;
}
```

包含头文件

定义数据变量

定义变量

调用STL中函数

循环输出

输出该容器内所有元素



图 16-7 程序中调用了 STL 中的函数实例

在如图 16-7 所示的代码中, 使用了 `push_back()`、`begin()` 和 `end()` 函数, 而这些函数在程序中并没有被定义过, 却可以使用, 这是因为这些函数已经被头文件 `vector.h` 所包含, 如同 `strcpy()` 函数被 `string.h` 头文件所包含一样。

【示例 16-2】该程序使用了 STL 的各种主要元素, 包括容器、迭代器、算法、函数对象。通过这个程序, 可以对 STL 的使用有一个整体的印象, 其实现代码及结果如图 16-8 所示。

```
#include <cstdlib>
#include <iostream>
#include <vector>

#include <algorithm>
using namespace std
template<class T>
struct Print
{
    void operator()(T& x) const
    {
        if( x % 2 == 0 )
        {
            cout<<x<<' ';
        }
    }
};

int main(int argc, char *argv[])
{
    // ... (rest of the code is not shown in the image)
}
```

包含容器 `vector` 的头文件

包含算法头文件

定义一个 `Print` 结构, 作为函数对象来使用

重载括号运算符

如果参数是偶数

输出参数



```

cout<<"--使用STL--"<<endl;
cout<<endl;
vector<int> vcInts;
for(int i = 0; i < 10; ++i)
{
    vcInts.push_back(i);
}
vector<int>::iterator iterBegin, iterEnd;
iterBegin = vcInts.begin();
iterEnd = vcInts.end();
cout<<"输出所有元素："<<endl;
for(; iterBegin != iterEnd; ++iterBegin)
{
    cout << *iterBegin << " ";
}
cout << endl;
cout<<"输出偶数元素："<<endl;

iterBegin = vcInts.begin();
for_each(iterBegin, iterEnd, Print<int>());
cout<<endl<<endl;
system("pause");
return 0;
}

```

实例化一个 vector 容器

插入到容器中

定义两个迭代器，分别用来指向容器的开始和结尾

指向容器的开始

指向容器的结尾

用迭代器遍历整个容器

输出容器中的各个元素

重新指向容器的开始

通过算法 for\_each 和函数对象 Print 输出容器中的元素

图 16-8 使用 STL 的各种主要元素实例



## 16.2 算法

STL 提供了大约 100 个实现算法的模板函数，用户通过调用一两个算法模板就可以完成所需要的功能，大大地提高了用户使用 C++ 进行程序设计的效率。

一般来说，STL 中的算法部分主要由头文件 `<algorithm>`、`<numeric>` 和 `<functional>` 组成。其中，头文件 `<algorithm>` 由一大堆模板函数组成，常用的函数涉及比较、交换、查找等。

STL 中包含的大量的常用算法，供程序员在需要时直接调用，只需知道该算法实现函数在哪个头文件下即可。

【示例 16-3】下面的程序使用了 STL 算法中一个最为常用的算法——排序，其实现代码及结果如图 16-9 所示。



```
#include <iostream>
#include <algorithm>
#include <functional>
#include <vector>
using namespace std

class myclass
{
public:
    myclass(int a, int b):first(a), second(b) { }
    int first;
    int second;
    bool operator < (const myclass &m) const
    {
        return first < m.first;
    }
};

bool less_second(const myclass & m1, const myclass & m2)
{
    return m1.second < m2.second;
}

int main()
{
    vector< myclass > vect;
    for(int i = 0 ; i < 3; i ++)
    {
        myclass my(10-i, i*3);
        vect.push_back(my);
    }
    for(i = 0 ; i < vect.size(); i ++)
        cout<<"("<<vect[i].first<<","<<vect[i].second<<")\n";
    sort(vect.begin(), vect.end());
    cout<<"after sorted by first"<<endl;
    for(i = 0 ; i < vect.size(); i ++)
        cout<<"("<<vect[i].first<<","<<vect[i].second<<")\n";

    cout<<"after sorted by second"<<endl;
    sort(vect.begin(), vect.end(), less_second);
    for( i = 0 ; i < vect.size(); i ++)
        cout<<"("<<vect[i].first<<","<<vect[i].second<<")\n";
    system("pause");
    return 0 ;
}
```

头文件 `algorithm` 含有算法相关函数

头文件 `vector` 含有向量相关函数

定义类

构造函数

重载运算符 `<`

根据第二个元素返回

创建对象

创建对象，并初始化

写入向量

调用排序算法

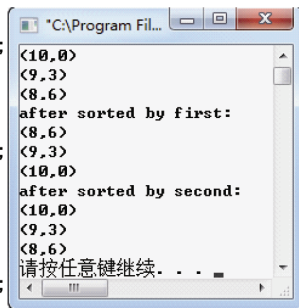


图 16-9 使用排序算法

在如图 16-9 所示的代码中，使用到了头文件 `<algorithm>` 和 `<functional>`，对输出的向量分别按照其第一个元素值和第二个元素值进行升序排列。可以看到，代码中并没有定义 `sort()` 函数，但在程序中两次调用了该函数，说明该函数已被头文件所包含。这两处调用的参数个数是不同的，说明该函数被重载了。



## 16.3 容器

在实际程序开发过程中，数据结构本身的重要性不亚于操作与数据结构的算法的重要性，当程序中存在对时间要求很高的部分时，数据结构的选择就显得更加重要了。STL 容器允许重复利用已有的实现构造自己特定类型下的数据结构，通过设置一些模板类，这些模板的参数允许用户指定容器中元素的数据类型，从而提高编程效率。





### 16.3.1 容器概述

容器部分主要由头文件<vector>、<list>、<deque>、<set>、<map>、<stack>和<queue>组成。常用的一些容器和容器适配器（可以看做由其他容器实现的容器）如表 16-1 所示。

表 16-1 容器与头文件的对应关系

数据结构	描 述	实现头文件
向量（vector）	连续存储的元素	<vector>
列表（list）	由节点组成的双向链表，每个节点包含一个元素	<list>
双队列（deque）	由连续存储的指向不同元素的指针所组成的数组	<deque>
集合（set）	由节点组成的红黑树，每个节点都包含着一个元素，节点之间以某种作用于元素对的谓词排列，没有两个不同的元素能够拥有相同的次序	<set>
多重集合（multiset）	允许存在两个次序相等的元素的集合	<set>
栈（stack）	后进先出的值的排列	<stack>
队列（queue）	先进先出的值的排列	<queue>
优先队列（priority_queue）	元素的次序是由作用于所存储的值对上的某种谓词决定的一种队列	<queue>
映射（map）	由{键,值}对组成的集合，以某种作用于键对上的谓词排列	<map>
多重映射（multimap）	允许键对有相等的次序的映射	<map>

STL 容器中容纳了大量的数据结构模板，前面使用的 vector 向量就是其中的一个模板，下面将详细介绍这些容器类。

### 16.3.2 向量

向量是一种 vector 容器类，在前面的程序中就引用过该类。向量就像是盛放变长数组的花园，所有 STL 容器中大约有一半是基于向量的。引入向量类的方式如图 16-10 所示。

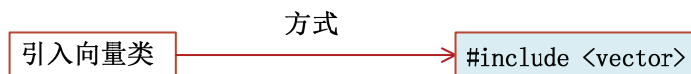


图 16-10 引入向量类的方式

事实上，vector 是一种动态数组，是基本数组的类模板，其内部定义了很多基本操作。既然这是一个类，那么就会有自己的构造函数。vector 类中定义了 4 种构造函数，如图 16-11 所示。

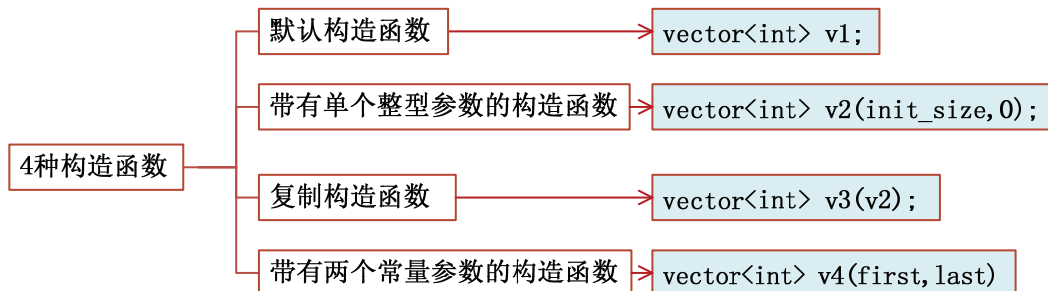


图 16-11 vector 类中定义的 4 种构造函数

此外，在实际程序中使用较多的是向量类的成员函数，其常用成员函数如图 16-12 所示。



向量类的常用成员函数

begin()、end()、push\_back()、  
insert()、assign()、front()、  
back()、erase()、empty()、  
at()、size()

图 16-12 向量类的成员函数

【示例 16-4】下面的程序是一个 Hello World 程序，其将一个字符串传送到一个字符向量中，然后每次显示向量中的一个字符，其实现代码及结果如图 16-13 所示。

```
#include <vector>
#include <iostream>
using namespace std;
char* szHW = "Hello World";
int main(int argc, char* argv[])
{
    vector<char> vec;
    vector<char>::iterator vi;

    char* cptr = szHW;
    while (*cptr != '\0')
    {
        vec.push_back(*cptr); cptr++;
    }
    for (vi=vec.begin(); vi!=vec.end(); vi++)
    {
        cout << *vi;
    }
    cout << endl;
    system("pause");
    return 0;
}
```

STL向量的头文件

一个字符数组，以“\0”结束

声明一个字符向量 vector

为字符数组定义一个游标 iterator

将一个指针指向 "Hello World" 字符串

push\_back()函数将数据放在向量的尾部

使用运算符 “\*” 将数据从游标指针中提取出来

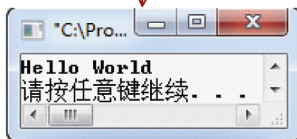


图 16-13 向量实例

### 16.3.3 列表

列表也是容器类中的一种，其所控制的长度为 N 的序列是以一个有着 N 个节点的双向链表来存储的，支持双向迭代器，其预编译头文件如图 16-14 所示。

头文件

列表预编译

#include <list>

图 16-14 列表预编译头文件

列表类的定义如图 16-15 所示。

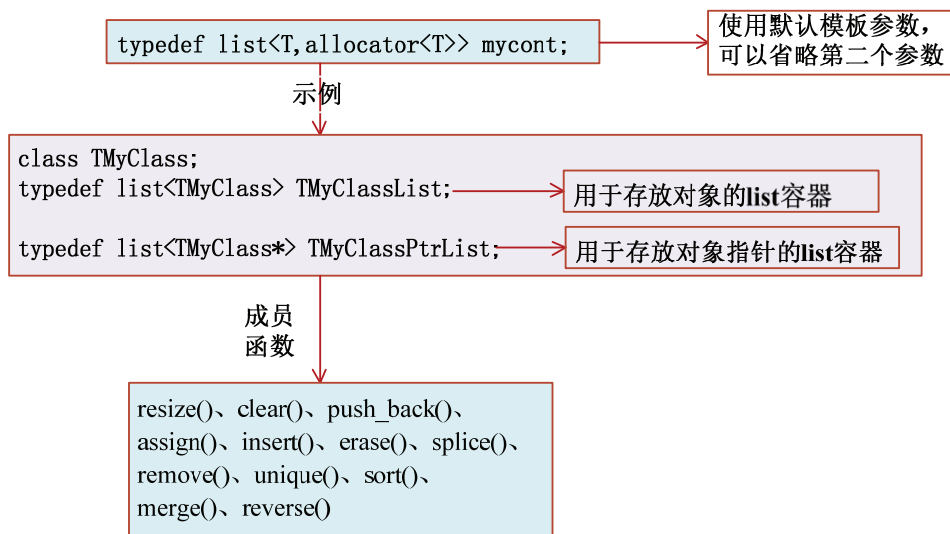


图 16-15 列表类的定义

【示例 16-5】下面的程序使用列表容器存储数据内容, 进行排序后输出, 其实现代码及结果如图 16-16 所示。

```
#include <list> → 包含列表类头文件
#include <iostream>
void main()
{
    using namespace std;
    list<int> c1; → 定义列表变量
    list<int>::iterator c1_Iter;
    c1.push_back(20);
    c1.push_back(10); → 调用函数写入列表
    c1.push_back(30);
    cout<<"Before sorting c1 =";
    for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
    cout << " " << *c1_Iter; → 输出列表中的内容
    cout << endl;
    c1.sort( ); → 调用列表类排序函数
    cout << "After sorting d =";
    for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
    cout << " " << *c1_Iter;
    cout << endl;
    c1.sort( greater<int>( ) ); → 调用降序排列函数
    cout << "After sorting with 'greater than' operation, c1 =";
    for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
    cout << " " << *c1_Iter;
    cout << endl;
}
```

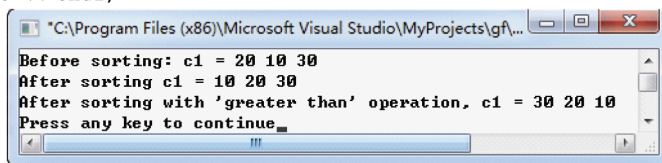


图 16-16 列表实例



### 16.3.4 集合

集合 (set) 也是容器的一种, 其特点如图 16-17 所示。

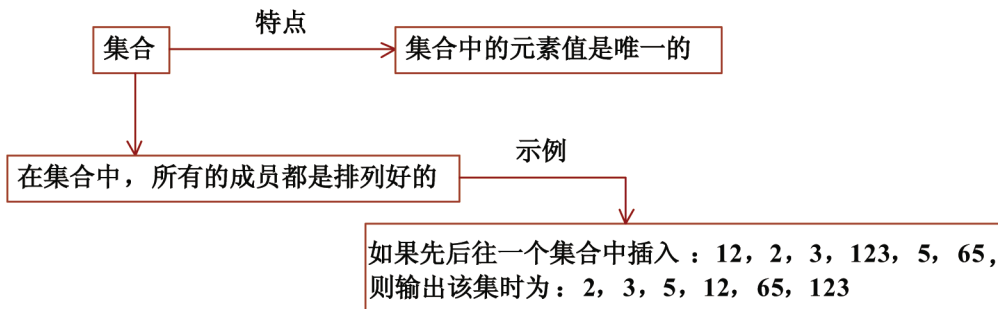


图 16-17 集合的特点

【示例 16-6】下面的程序使用集合存储数据内容, 将值插入到集合后输出, 其实现代码及结果如图 16-18 所示。

```
#include <string>
#include <set>
#include <iostream>
using namespace std;
int main(int argc, char* argv[])
{
    set <string> strset;
    set <string>::iterator si;
    strset.insert("cantaloupes");
    strset.insert("apple");
    strset.insert("orange");
    strset.insert("banana");
    strset.insert("grapes");
    strset.insert("grapes");
    for (si=strset.begin(); si!=strset.end(); si++)
    {
        cout<< *si << " ";
    }
    cout << endl;
    return 0;
}
```

包含集合类头文件

定义变量

调用函数插入数据到集合中

循环输出集合中数据

图 16-18 集合实例

### 16.3.5 双端队列

双端队列是一个 queue 容器类 (即队列容器), 其预编译头文件如图 16-19 所示。



图 16-19 队列预编译头文件



【示例 16-7】下面的程序使用双端队列存储数据内容，对该队列进行插入和删除元素操作。其实现代码及结果如图 16-20 所示。

```
#include <iostream>
#include <deque>
using namespace std
typedef deque<int> INTDEQUE;
void put_deque(INTDEQUE deque, char *name)
{
    INTDEQUE::iterator pdeque;
    cout << "The contents of" << name << " : ";
    for(pdeque = deque.begin(); pdeque != deque.end(); pdeque++)
        cout << *pdeque << " ";
    cout<<endl;
}
void main(void)
{
    INTDEQUE deq1;

    INTDEQUE::iterator i;
    put_deque(deq1,"deq1");
    deq1.push_back(2);
    deq1.push_back(4);
    cout<<"deq1.push_back(2) and deq1.push_back(4):"<<endl;
    put_deque(deq1,"deq1");
    deq1.push_front(5);
    deq1.push_front(7);
    cout<<"deq1.push_front(5) and deq1.push_front(7):"<<endl;
    put_deque(deq1,"deq1");
    deq1.insert(deq1.begin()+1, 3, 9);
    cout<<"deq1.insert(deq1.begin()+1, 3, 9):"<<endl;
    put_deque(deq1,"deq1");
    cout<<"deq1.at(4)="<<deq1.at(4)<<endl;
    cout<<"deq1[4]="<<deq1[4]<<endl;
    deq1.at(1)=10;
    deq1[2]=12;
    cout<<"deq1.at(1)=10 and deq1[2]=12 :"<<endl;
    put_deque(deq1,"deq1");
    deq1.pop_front();
    deq1.pop_back();
    cout<<"deq1.pop_front() and deq1.pop_back():"<<endl;
    put_deque(deq1,"deq1");
    deq1.erase(deq1.begin()+1);
    cout<<"deq1.erase(deq1.begin()+1):"<<endl;
    put_deque(deq1,"deq1");
}
```

定义双端队列容器

从前向后显示 deque 队列的全部元素

使用迭代器输出

定义双端队列 deq1 对象初始为空

声明一个名为 i 的双向迭代器变量

```
The contents of deq1 :
deq1.push_back(2) and deq1.push_back(4):
The contents of deq1 : 2 4
deq1.push_front(5) and deq1.push_front(7):
The contents of deq1 : 7 5 2 4
deq1.insert(deq1.begin()+1,3,9):
The contents of deq1 : 7 9 9 5 2 4
deq1.at(4)=5
deq1[4]=5
deq1.at(1)=10 and deq1[2]=12 :
The contents of deq1 : 7 10 12 9 5 2 4
deq1.pop_front() and deq1.pop_back():
The contents of deq1 : 10 12 9 5 2
deq1.erase(deq1.begin()+1):
The contents of deq1 : 10 9 5 2
Press any key to continue
```

图 16-20 队列实例

图 16-20 中的代码演示了 deque 如何进行插入、删除等操作，双端队列 deq1 首先为空，通过函数 push\_front() 和函数 push\_back() 分别在该队列的前端和后端插入元素，通过函数 insert() 在队列的中间插入元素，通过函数 erase() 在队列中删除元素。



### 16.3.6 栈

栈 (stack) 是一种特殊的容器，其特征是后进先出，即先进来的元素放在栈底，最后才能取出。栈容器支持的操作有 5 种，如图 16-21 所示。

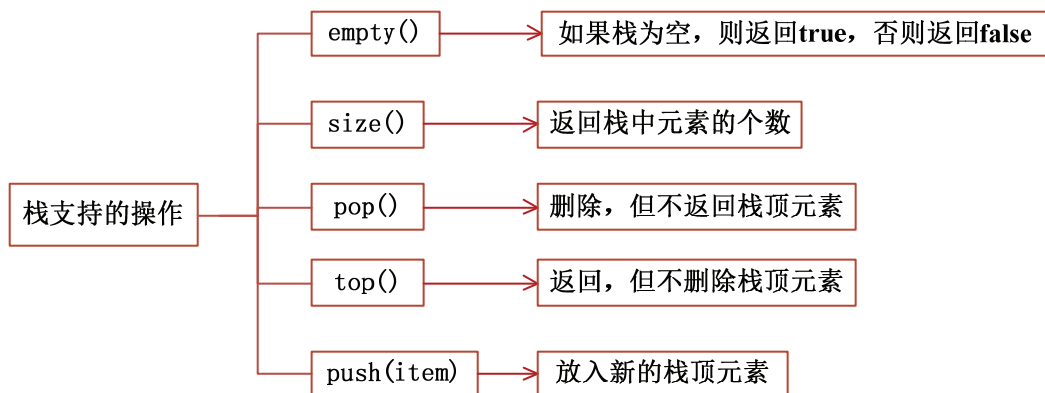


图 16-21 栈容器支持的操作

【示例 16-8】下面的程序实现了将 10 个元素放入栈中，然后输出，仔细观察其输出是否有变化。其实现代码及结果如图 16-22 所示。

```
#include <stack>
#include <iostream>
using namespace std
int main(int argc, char* argv[])
{
    const int ia_size = 10;
    int ia[ia_size] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    int ix = 0;
    stack<int> intStack;
    for ( ; ix < ia_size; ++ ix)
    {
        intStack.push(ia[ix]);
    }
    if (intStack.size() != ia_size)
    {
        cout << "error!" << endl;
        return -1;
    }
    int value;
    while (!intStack.empty())
    {
        value = intStack.top();
        cout<<value<<" ";
        intStack.pop();
    }
    cout<<endl;
    return 0;
}
```

图 16-22 栈实例



### 16.3.7 映射和多重映射

映射和多重映射用于对数据进行快速和高效的检索。同样，在程序中使用映射和多重映射容器也需添加头文件，如图 16-23 所示。

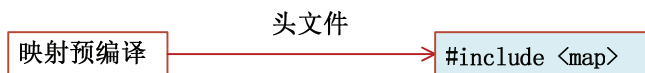


图 16-23 映射头文件

此外，映射 map 支持下标运算符 operator[]，可以用访问普通数组的方式访问 map，或下标为 map 的键。而在 multimap 中一个键可以对应多个不同的值。

【示例 16-9】下面的程序使用映射容器存储数据内容，并调用其函数完成元素键值的检索功能。其实现代码及结果如图 16-24 所示。

```
#include <iostream>
#include <map>
using namespace std;
int main(void)
{
    map<char, int, less<char> > map1;
    map<char, int, less<char> >::iterator mapIter;
    map1['c']=3;
    map1['d']=4;
    map1['a']=1;
    map1['b']=2;
    for(mapIter=map1.begin(); mapIter!=map1.end(); ++mapIter)
        cout<<" "<<(*mapIter).first<<": "<<(*mapIter).second;
    cout<<endl;
    map<char, int, less<char> >::const_iterator ptr;
    ptr=map1.find('d');
    cout<<" "<<ptr->first<<"键对应于值: "<<ptr->second<<endl;
    return 0;
}
```

包含映射头文件

定义变量

char是键的类型，int是值的类型

first对应定义中的char键，second对应定义中的int值

键对应于值

图 16-24 映射实例

图 16-24 中的程序说明了 map 中键与值的关系。代码中，首先定义了映射容器，其键的类型为字符型，值的类型为整型。对该容器 map1 进行了初始化，在其中存储 4 个字符对应的整型数值并输出。再通过函数 find() 找出其中 d 键对应的值。



## 16.4 迭代器

迭代器实际上是一种泛化指针，如果一个迭代器指向了容器中的某一成员，那么迭代器将通过自增自减来遍历容器中的所有成员。迭代器是联系容器和算法的媒介，是算法操作容器的接口，如图 16-25 所示。



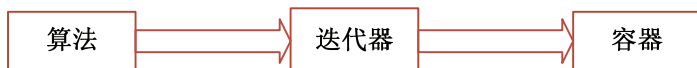


图 16-25 迭代器的作用

简单来说, 几乎 STL 提供的所有算法都是通过迭代器存取元素序列进行工作的, 每个容器都定义了它本身所专有的迭代器, 用以存取容器中的元素。

STL 中的迭代器主要由头文件<utility>、<iterator>和<memory>组成, 如图 16-26 所示。

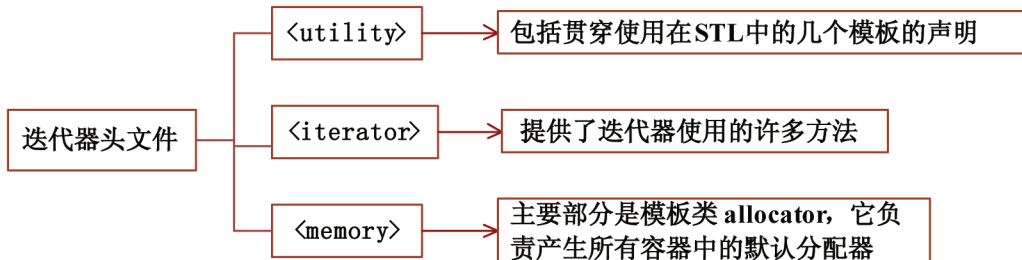


图 16-26 STL 中的迭代器的头文件

【示例 16-10】下面的程序使用输入/输出迭代器实现了将一个文件输出到屏幕的功能, 实现代码及结果如图 16-27 所示。

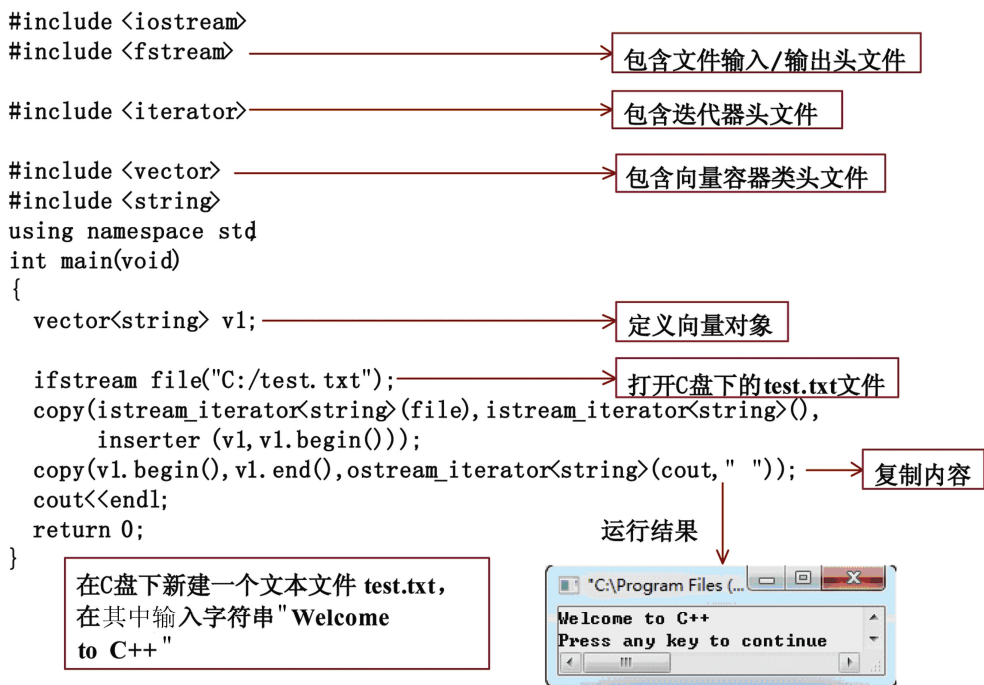


图 16-27 输入/输出迭代器

在如图 16-27 所示的代码中, 用到了输入迭代器 `istream_iterator` 和输出迭代器 `ostream_iterator`。程序实现了将一个文件输出到屏幕的功能, 先将文件读入, 然后通过输入迭代器把文件内容复制到类型为字符串的向量容器内, 最后由输出迭代器输出。





## 16.5 小结

本章主要介绍了标准模板库 STL 的相关内容。首先对 STL 的概念及其在程序设计中的重要性做了概括介绍，并通过一个具体示例引入了 STL 的应用。本章主要讲解了 STL 的几个组成部分，包括算法、容器、迭代器等，对于每个组成部分在具体程序中的使用，都通过一个实例来讲解，读者仔细理解这些实例即可理解 STL 的优势。

## 16.6 习题

【题目 16-1】使用 vector 容器，进行数据动态增加和删除，并输出中间结果。

【题目分析】本题主要考查 STL 中的 vector 和迭代器的相关知识，重点是掌握迭代器的使用。

【关键代码】

```
void disp(vector<int>& x)
{
    unsigned int i=0;
    for(;i<x.size();i++)
    {
        cout<<x[i]<<" ";
    }
}

vector<int> obD(5,0);
vector<int>::iterator pD=obD.end();
pD=obD.insert(pD,1);
disp(obD);
cout<<endl;
obD.insert(pD,2,3);
disp(obD);
iter=obD.erase(iter);
disp(obD);
cout<<endl;
obD.erase(iter,obD.end());
disp(obD);
cout<<endl;
```

【题目 16-2】定义 map 和 multimap 容器，并插入数据和输出结果。

【题目分析】本题主要考查 STL 中 map 和 multimap 容器的相关知识，重点是掌握 map 和 multimap 容器的使用。

【关键代码】

```
pair<int,string> sz[2]={pair<int,string>(1,"A"),pair<int,string>(2,"B")};
pair<int,string> t(2,"X");
map<int,string>obM(sz,sz+2);
map<int,string>::iterator itM=obM.begin();
pair<map<int,string>::iterator,bool> res=obM.insert(t);
if(res.second)
{
    cout<<"插入成功"<<endl;
}
else
{
}
```



```
        cout<<"以包含关键字与 t 相同的元素: "<<(*res.first).second<<endl;
    }
    multimap<int,string> obDM(sz,sz+2);
    multimap<int,string>::iterator itDM=obDM.begin();
    itDM=obDM.insert(t);
    cout<<"插入的元素为: "<<(*itDM).second<<endl;
```

# 第 17 章 程序调试与异常处理

在程序设计的过程中，会不可避免地出现各种错误。随着程序规模的扩大，程序中出现的错误也会越来越多。为此，各种程序设计语言都需要进行程序调试。此外，异常处理是 C++ 的一个主要特征，它提供了完美的结果出错处理方法。本章将简要介绍程序调试与异常处理的相关知识。



## 17.1 程序错误

使用任何一种语言设计的程序都或多或少存在程序错误，程序错误常常是由于程序设计人员的疏忽而产生的。一般而言，程序错误主要包含编译错误、逻辑错误和运行错误 3 类。

### 17.1.1 编译错误

编译错误也称为语法错误，其定义如图 17-1 所示。

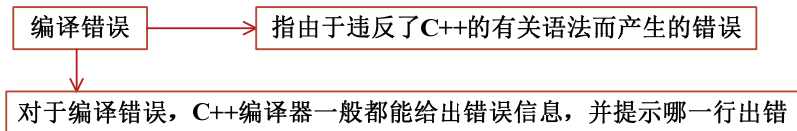


图 17-1 编译错误

对于初学者来说，编译错误是最容易犯的一类错误，但也是最容易排除的一类错误。

【示例 17-1】下面的程序在屏幕上输出“Welcome to C++”字样，代码如图 17-2 所示。

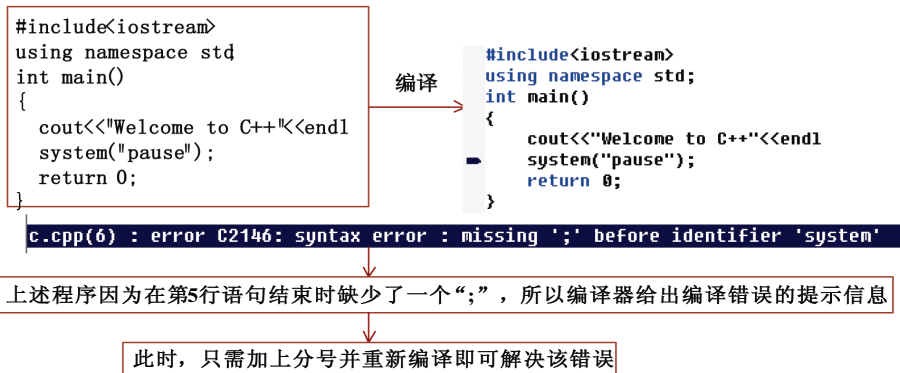


图 17-2 编译错误实例



```
#include<iostream.h>
int main(void)
{
    int x;
    int y
    double z;
    y=w+x;
}
error C2144: syntax error : missing ';' before type 'double'
error C2065: 'w' : undeclared identifier
```



**提示：**（1）由于 C++ 是区分大小写的，因此在书写标识符时，如果忽略了其大小写，那么编译系统也将给出相应错误提示。

（2）在保留字和标识符中间不能有空格。

（3）语句的末尾必须加分号，作为语句结束符。

### 17.1.2 逻辑错误

逻辑错误是指程序存在逻辑上的缺陷，因此当程序运行后，得不到所期望的结果。逻辑错误并不直接导致程序在编译期间和运行期间出现错误，因此不像编译错误那么容易发现。

逻辑错误的产生如图 17-3 所示。

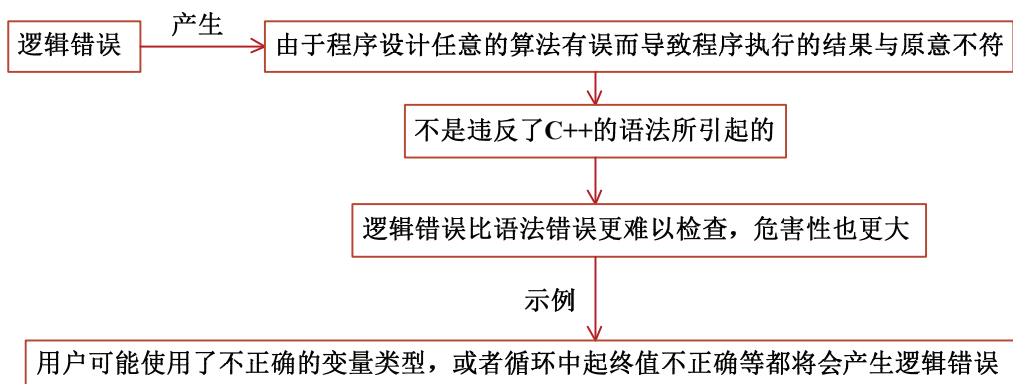


图 17-3 逻辑错误

一般来说，逻辑错误不会产生错误提示信息，需要用户仔细分析程序。

**【示例 17-2】**下面的程序实现一个冒泡排序函数，对用户定义的一个数组进行升序排列后输出。其实现代码及结果如图 17-4 所示。



```
#include <iostream>
using namespace std
int sort(int array[ ],int n)
{
    int temp;
    for (int i=1;i<n;i++)
    {
        for (int j=n-1;j>1;j--)
        {
            if (array[j]<array[j-1])
            {
                temp=array[j-1];
                array[j-1]=array[j];
                array[j]=temp;
            }
        }
    }
    return 0;
}
```

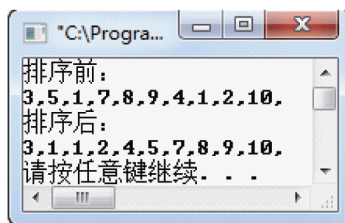
定义排序函数

逻辑错误：下标取值不正确

交换数组元素

```
int main()
{
    int array[10]={3,5,1,7,8,9,4,1,2,10};
    int i;
    cout<<"排序前: "<<endl;
    for (i=0;i<10;i++)
    cout<<array[i]<<" ";
    cout<<endl;
    sort(array,10);
    cout<<"排序后: "<<endl;
    for (i=0;i<10;i++)
    cout<<array[i]<<" ";
    cout<<endl;
    system("pause");
    return 0;
}
```

定义一维数组并初始化



程序的运行结果并没有按照读者的意图对数组中的元素进行升序排列

说明冒泡排序函数存在逻辑错误

逻辑错误的原因在于第 8 行的 for 循环语句中变量 j 的终值应为 1，即将语句 j&gt;1 改为 j&gt;=1

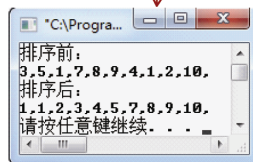


图 17-4 逻辑错误实例



**提示：**逻辑错误的排除需要读者仔细理解算法，一定要注意数组的下标越界、内存溢出等问题。

### 17.1.3 运行错误

运行错误可以认为是逻辑错误的一种特殊情况，其定义如图 17-5 所示。

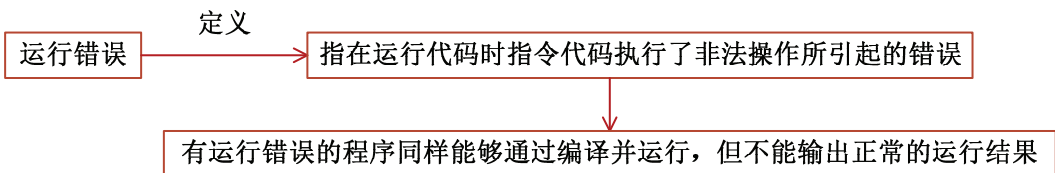


图 17-5 运行错误

一般来说，编译器对许多错误将以运行错误的形式体现，如分母为零、被操作的驱动器未准备好或磁盘读/写有错等，导致程序不能继续执行。

【示例 17-3】下面的程序对一个除数为 0 的变量进行了除法运算，这是不允许的。C++将以运行错误的形式体现该错误，实现代码及结果如图 17-6 所示。

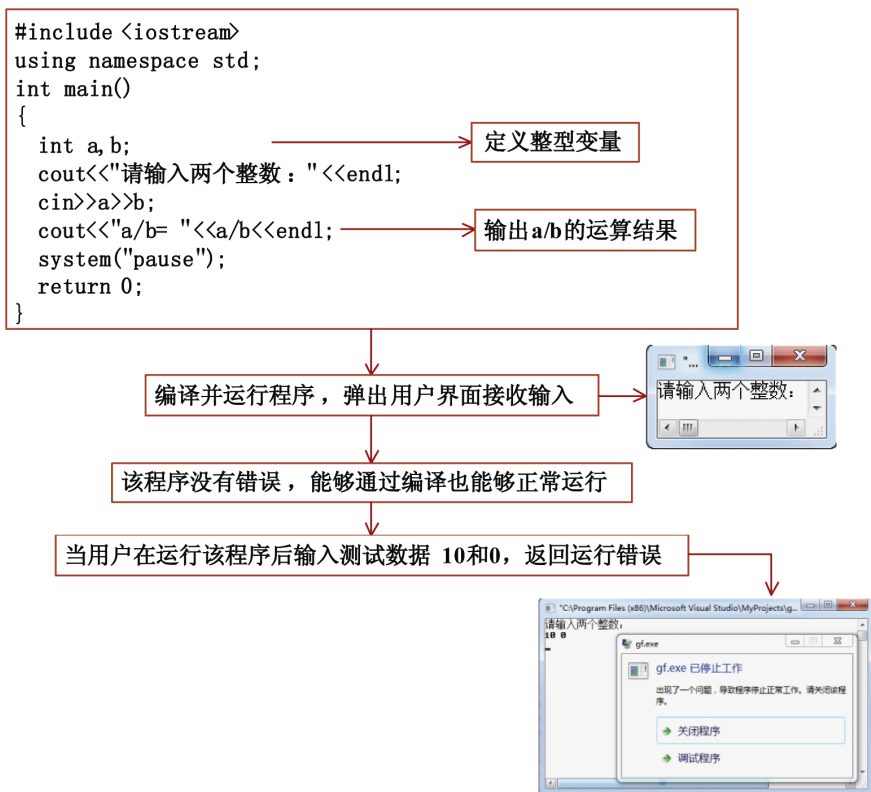


图 17-6 运行错误实例

因此，在判断一个程序是否有运行错误时不能单靠 C++语法的检查，还需要加强程序健壮性检查，从而避免出现运行错误。

#### 17.1.4 程序调试

程序调试是指发现程序的错误及其对错误的改正。程序调试不能仅依赖于编译器发现错误，更重要的是程序设计人员对算法和 C++语法规则的理解。

一般来说，C++源程序出现编译错误后，编译器将自动显示出错行和错误提示，读者根据提示进行修改即可，编译错误一般不需要进行逐条语句调试。

下面介绍一下断言函数（assert）。



断言就是判断，主要有两种，`assert` 和 `ASSERT`。其中，`assert` 是标准 C++ 中的宏，`assert` 宏的原型定义在 `<assert.h>` 中；`ASSERT` 是 MFC 中的宏，它们的使用方法一样。使用格式如下：

```
assert(expression);
```

执行时先测试逻辑表达式 `expression`，若 `expression` 的值为 0，则中断执行并打印一段说明消息；否则执行 `assert` 后的语句。



## 17.2 异常处理

异常是指程序在运行过程中，由于使用环境的变化及用户的操作而产生的错误，因此异常可以认为是错误的狭义概念。异常处理是在程序设计过程中，针对可预测的异常编制相应的预防代码或处理代码，以便防止异常发生后造成严重后果。

### 17.2.1 基本思想

一个应用程序，既要保证其正确性，还应有容错能力，如图 17-7 所示。

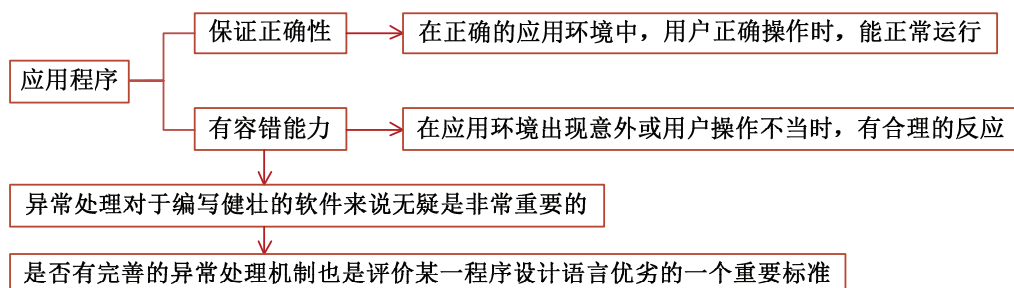


图 17-7 应用程序

一般而言，异常处理的基本思想是：在底层发生的问题，逐级上报，直到有能力可以处理异常的那级为止，如图 17-8 所示。

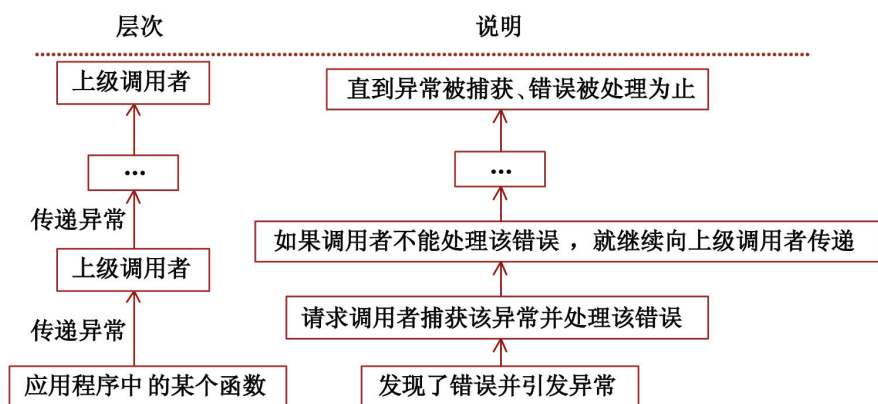


图 17-8 异常处理的基本思想

所有的程序设计语言对于异常处理的思想基本上都是类似的，就是捕获异常后进行处理。如果程序最终没有相应的代码处理该异常，那么该异常最后被 C++ 系统所接受，C++ 系统就简单地终止程序运行，异常的传递如图 17-9 所示。

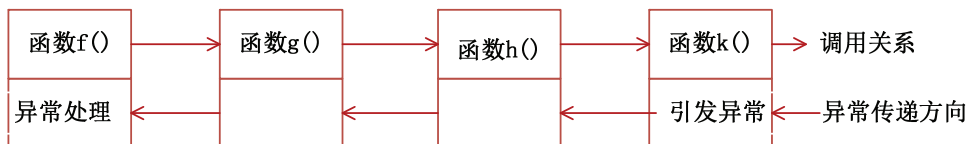


图 17-9 异常的传递方向

C++异常处理的目的是在异常发生时，尽可能地减少破坏，使其不影响或尽量少影响程序其他部分的运行。

【示例 17-4】在前面提到的除数为 0 的示例，如果当用户输入的第二个整数为 0 时程序将出现异常退出。为了增强该程序的健壮性，可以通过异常处理的基本思想进行处理，为其加上输入的限制，将该程序改写，其实现代码及结果如图 17-10 所示。

```
#include <iostream>
using namespace std;
int main()
{
    int a, b;
    cout<<"请输入两个整数："<<endl;
    cin>>a>>b;
    if (b==0)
    {
        cout<<"除数不能为零，请重新输入除数"<<endl;
        cin>>b;
        cout<<"a/b= "<<a/b<<endl;
    }
    else
    {
        cout<<"a/b= "<<a/b<<endl;
        system("pause");
        return 0;
    }
}
```

判断除数是否为 0

输入为0则重新输入

输出a/b的结果

图 17-10 异常处理

## 17.2.2 抛出异常

如果程序发生异常情况，而在当前环境中获取不到异常处理的足够信息，可以将异常抛出。抛出异常的含义，如图 17-11 所示。

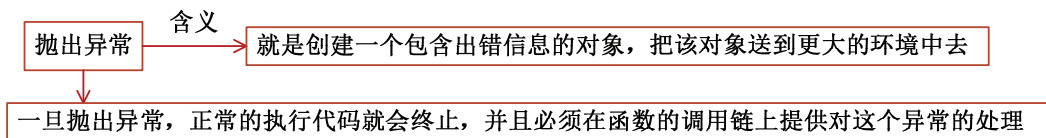


图 17-11 抛出异常的含义

当一段程序中发现错误数据，但是该程序不知道如何处理时，可以抛出异常。在 C++中，使用 `throw` 来抛出异常。抛出异常的语法如图 17-12 所示。



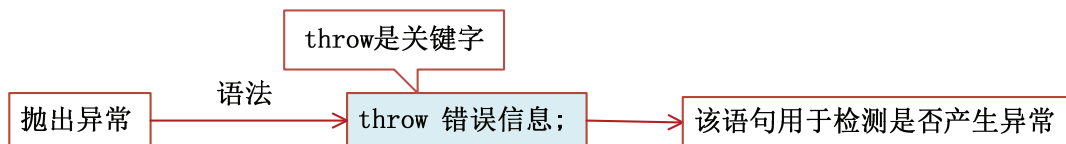


图 17-12 抛出异常的语法

【示例 17-5】将上述除数为 0 的程序进行适当处理，当用户输入除数为 0 时抛出异常，其实现代码及结果如图 17-13 所示。

```

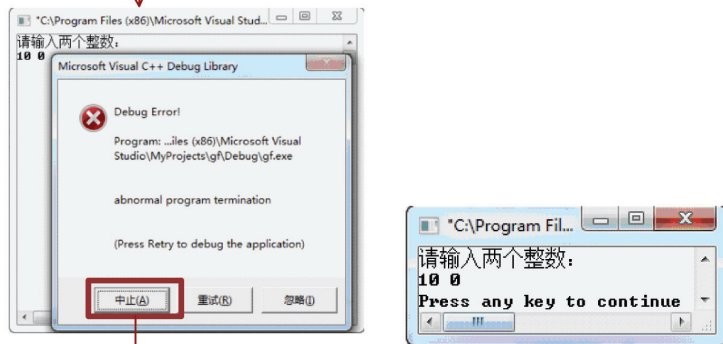
#include <iostream>
using namespace std;
int main()
{
    int a,b;
    cout<<"请输入两个整数："<<endl;
    cin>>a>>b;
    if (b==0)
    {
        throw b;
    }
    else
    {
        cout<<"a/b= "<<a/b<<endl;
        system("pause");
        return 0;
    }
}
  
```

判断除数是否为 0

抛出异常

输出 a/b 的结果

输入两测试数据 10 和 0，程序将退出执行



代码只简单地在函数中抛出异常，不进行异常处理，因此程序会中途退出

图 17-13 抛出异常

当用户使用 throw 语句抛出异常时，可以避免程序出现运行错误。但是，如果只有异常的抛出，程序将中途退出，这也是在具体程序设计中不允许的，为此，C++引入了异常的捕获和处理机制。

### 17.2.3 捕获异常

如果一个函数抛出一个异常，它必须假定该异常能被捕获和处理。正如前面提到的，允许对一个问题集中在一处解决，这也正是 C++语言异常处理的一个优点。

在 C++中，提供了语句 try...catch 来捕获异常，如图 17-14 所示。

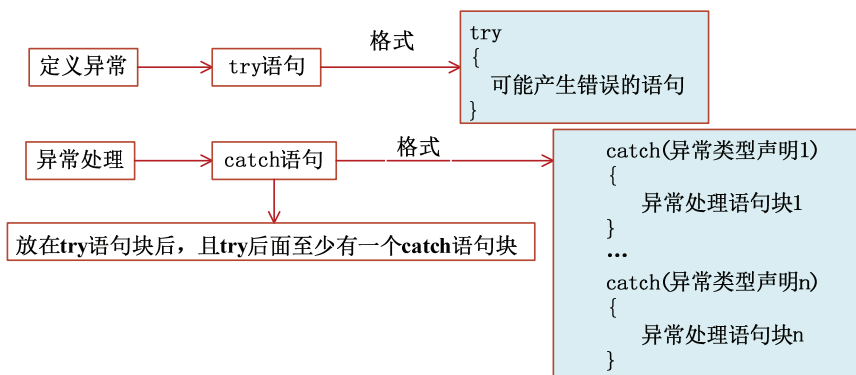


图 17-14 捕获异常的语句

一旦某个异常被抛出, 异常处理机制将会按照书写的 `catch` 语句块的代码顺序依次寻找匹配的语句。一旦找到一个相匹配语句, 则像调用函数一样进入到该处理器中进行处理。

在查找匹配 `catch` 期间, 找到的 `catch` 不必是与异常最匹配的那个 `catch`, 相反会选中第一个找到的可以处理该异常的 `catch`。因此, 在 `catch` 子句列表中, 最特殊的 `catch` 必须最先出现。

【示例 17-6】下面是关于异常抛出和捕获的简单程序, 在函数中抛出一个整型异常, 在主函数中捕获这个异常, 实现代码及结果如图 17-15 所示。

```
#include <iostream>
using namespace std;

void func1( int n )
{
    if( n == 1 )
    {
        cout<<"func1 抛出异常"<<endl;
        throw 1;
    }
}

void func2(int n)
{
    if( n == 2 )
    {
        cout<<"func2 抛出异常"<<endl;
        throw 2;
    }
}

int main()
{
    for( int i = 1; i < 3; i ++ )
    {
        try{
            func2( i );
        }
        catch( int )
        {
            cout<<"捕捉到int类型的异常"<<endl;
        }
    }
    system("pause");
    return 0;
}
```

定义func1()函数

参数n的值为1时

抛出异常

定义func2()函数

参数n的值为2时

抛出异常

捕获异常

异常处理

图 17-15 关于异常抛出和捕获的简单程序



在如图 17-15 所示的程序中，函数 func1() 和 func2() 都可以抛出 int 类型的异常，因此只有一个 int 类型的异常处理。整个 try 块在 for 循环之内，因此异常的抛出不影响 for 循环的运行。

### 17.2.4 自定义异常对象

抛出类型可以是自定义的类对象，这样做的优点是，一方面可以使用不同的异常类型来区分不同函数在不同情况下引发的异常；另一方面，自定义的类对象可以用来传递信息，例如：

```
#include <iostream>
using namespace std;
class Exception1
{
public:
    void disp()
    {
        cout<<"Exception1 异常被引发"<<endl;
    }
};
class Exception2
{
public:
    Exception2(const Exception1&)           // 以 Exception1 来复制构造
{
    }
    void disp()
    {
        cout<<"Exception2 异常被引发"<<endl;
    }
};
void fun()
{
    throw Exception1();
}
int main()
{
    try
    {
        fun();
    }
    catch(Exception2& Ex)
    {
        Ex.disp();
    }
    catch(Exception1& Ex)
    {
        Ex.disp();
    }
    return 0;
}
```

输出结果：

```
Exception1 异常被引发
Press any key to continue
```

代码解析：首先定义了两个类 Exception1 和 Exception2，函数 fun 不进行任何操作，仅仅是抛出了 Exception1 类的对象，在对象创建时，使用的还是系统默认的非参构造函数。在 main 函数中有两个 catch 来捕获异常，前一个用以匹配 Exception2 类型，后一个用以匹配 Exception1 类型。这里有一个疑问是，在代码第 14 行 Exception2 类中定义了一个构造函数，其参数是



Exception1 类对象,那么异常处理机制是否会自动将 fun 函数中抛出的 Exception 对象转换成一个 Exception2 对象,使得第一个 catch 被匹配呢?答案是否定的,异常处理机制在处理异常的过程中不会做任何形式的转换,只寻找和参数最匹配的 catch 进行处理,异常将第二个 catch 语句捕获,当然有特殊情况,那就是类继承。



## 17.3 异常处理实例

C++异常处理机制是一个用来有效处理运行错误的非常强大且灵活的工具,它提供了更多的弹性、安全性和稳固性,克服了传统方法所带来的问题。事实上,C++中的异常处理机制是一种把控制权从异常发生的地点转移到一个匹配的处理函数或功能块的机制。

【示例 17-7】下面的程序求一元二次方程的实根,要求加上异常处理,判断  $b^2-4ac$  是否大于 0,成立则求两个实根,否则要求重新输入。其实现代码及结果如图 17-16 所示。

```
#include <iostream>
#include <math.h>
using namespace std;
double sqrt_delta(double d)
{
    if(d < 0)
        throw 1;
    return sqrt(d);
}

double delta(double a, double b, double c)
{
    double d = b * b - 4 * a * c;
    return sqrt_delta(d);
}

int main()
{
    double a, b, c;
    cout << "请输入a,b,c的值" << endl;
    cin >> a >> b >> c;
    while(true)
    {
        try
        {
            double d = delta(a, b, c);
            cout << "x1: " << (d - b) / (2 * a) << endl;
            cout << "x2: " << -(b + d) / (2 * a) << endl;
            break;
        }
        catch(int)
        {
            cout << "delta < 0, 请重新输入a, b, c.";
            cin >> a >> b >> c;
        }
    }
    system("pause");
    return 0;
}
```

包含头文件

定义函数

抛出异常

定义函数

调用sqrt\_delta()函数

循环

定义异常

调用函数

定义异常处理

输入两组测试数据

图 17-16 异常处理实例



上述程序段实现了对于用户输入的一元二次方程系数的判断，由于只有方程的系数符合  $b^2 - 4ac > 0$  的条件时才有实根，所以上述代码中的 `sqrt_delta()` 函数中包含了异常定义及对异常的处理。



**注意：** `catch` 语句后的参数（即数据类型）需要与 `throw` 语句后的表达式数据类型相同。



## 17.4 小结

本章主要讲解了 C++ 程序调试和异常处理的基本内容，主要包括 C++ 程序中的常见错误：编译错误、逻辑错误和运行错误。此外，异常处理是程序设计语言的一个重要组成部分，C++ 的异常处理机制主要由定义异常、定义异常处理和 `throw` 语句等组成。C++ 中处理异常的语句主要包括 `try` 语句、`catch` 语句和 `throw` 语句。



## 17.5 习题

【题目 17-1】自定义一个对于除法中除数为 0 或者不是数字的异常抛出，并在上级函数中进行捕获，最后输出异常结果。程序运行结果如图 17-17 所示。

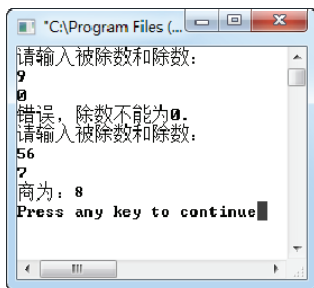


图 17-17 运行结果

【题目分析】本题主要考查读者对异常机制处理的相关知识的理解，重点是掌握异常机制的概念及其使用。

### 【关键代码】

```
int calc()
{
    cout<<"请输入被除数和除数: "<<endl;
    int n=0;
    int m=0;
    cin>>n;
    cin>>m;
    if(!cin.good())
    {
        cin.clear();
        cin.sync();
        throw "错误，输入的不是数字.";
    }
    if(m==0)
    {
        throw "错误，除数不能为0.";
    }
}
```



```
    }  
    return n/m;  
}  
int main()  
{  
    int res=0;  
    while(true)  
    {  
        try  
        {  
            res=calc();  
        }  
        catch(const char* s)  
        {  
            cout<<s<<endl;  
            continue;  
        }  
        cout<<"商为: ";  
        cout<<res<<endl;  
        break;  
    }  
    return 0;  
}
```

【题目 17-2】自定义一个求平方根的函数，该函数有抛出异常的处理能力，程序运行结果如图 17-18 所示。

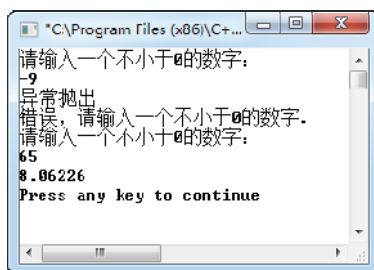


图 17-18 运行结果

【题目分析】本题主要考查读者对异常机制处理的相关知识的理解，重点是掌握异常机制的概念及其使用。

【关键代码】

```
double calc()  
{  
    cout<<"请输入一个不小于 0 的数字: "<<endl;  
    double num=-1;  
    cin>>num;  
    if(!cin.good())  
    {  
        cin.clear();  
        cin.sync();  
        throw "错误, 输入的不是数字.";  
    }  
    if(num<0)  
    {  
        throw "错误, 请输入一个不小于 0 的数字.";  
    }  
    return sqrt(num);  
}  
void keepInput()
```



```
{
    double res=0;
    try
    {
        res=calc();
    }
    catch(...)
    {
        cout<<"异常抛出"<<endl;
        throw;
    }
    cout<<res<<endl;
}

int main()
{
    while(true)
    {
        try
        {
            keepInput();
        }
        catch(const char* s)
        {
            cout<<s<<endl;
            continue;
        }
        break;
    }
    return 0;
}
```

# 第 18 章 文件

C++提供了专门的处理文件输入/输出的类。利用这些类进行文件操作更加灵活，功能更加强大。这些类包括用于输出的 `ofstream` 类、用于输入的 `ifstream` 类，以及既可用于输出也可以用于输出的 `fstream` 类。

## 18.1 文件概述

C++的文件把数据看做是一连串的字符流或二进制流，称为流式文件，增加了处理的灵活性。在程序设计中，使用文件这一概念是出于三个方面的考虑，如图 18-1 所示。

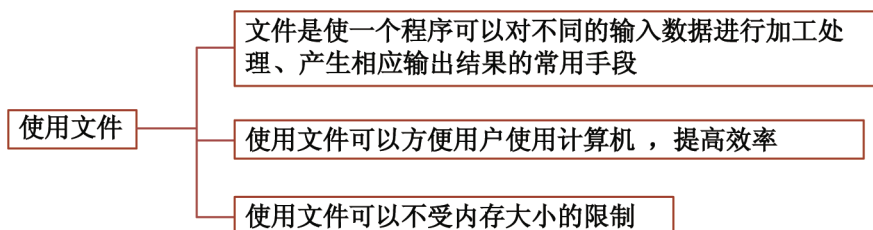


图 18-1 使用文件这一概念是出于三个方面的考虑

在计算机中，文件根据分类标准的不同可分为不同类型的文件。如图 18-2 所示。

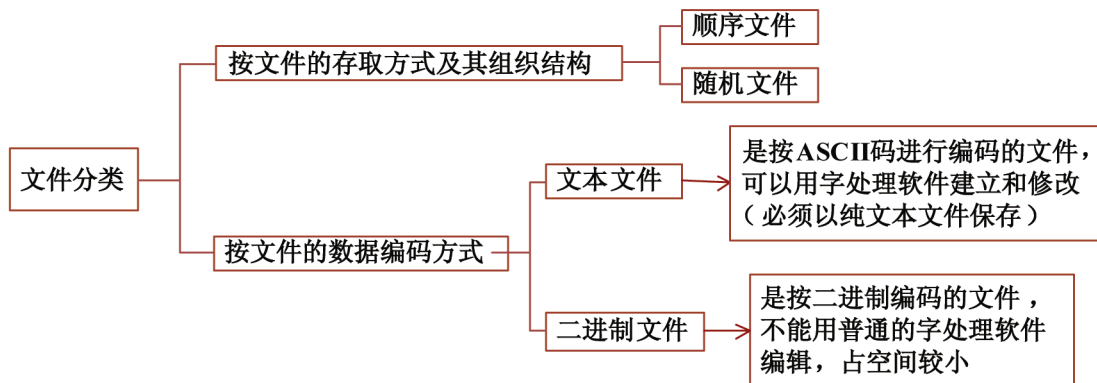


图 18-2 文件的分类

顺序文件与随机文件的区别及其优缺点如图 18-3 所示。



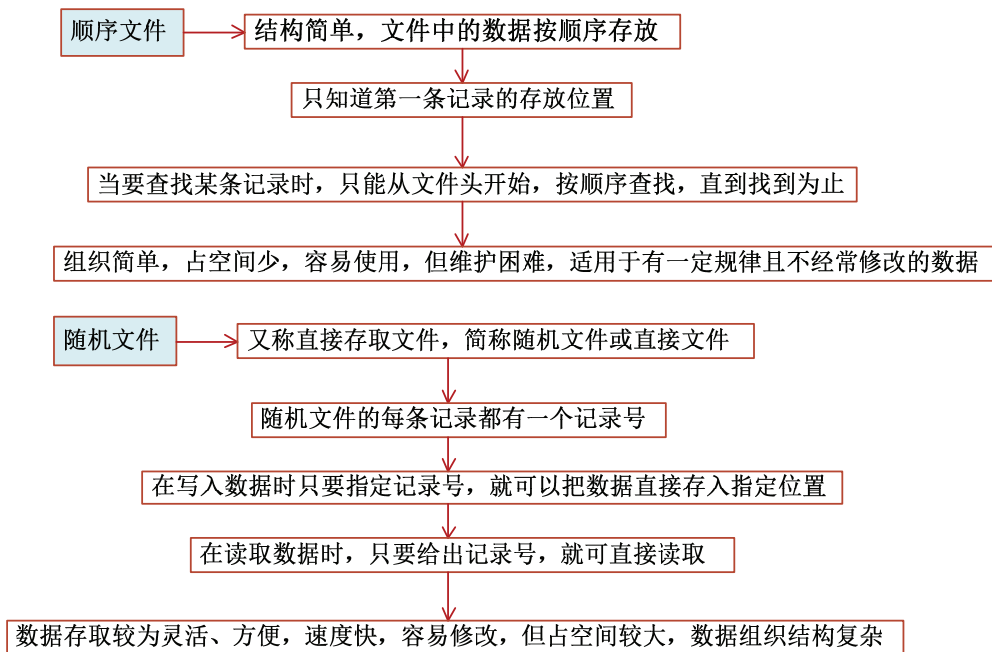


图 18-3 顺序文件与随机文件的区别及其优缺点

### 18.1.1 操作文件的过程

所谓文件，其实是一个独立的存储数据的单位，通常用一个名字（文件名）来表示。不同于内存中的存储单位，文件是可以永久保存的。即便是计算机断电，文件仍然可以存在。通常，文件可以存储于磁盘、磁带、光盘、闪存盘等各种介质上，只要有相关的设备支持存取即可。

一般来说，操作文件就是通过程序将文件读入到内存，根据计算序列再将其中的数据转入寄存器，然后由 CPU 进行处理。最终，根据需要由程序将数据写回到文件中，其处理过程如图 18-4 所示。

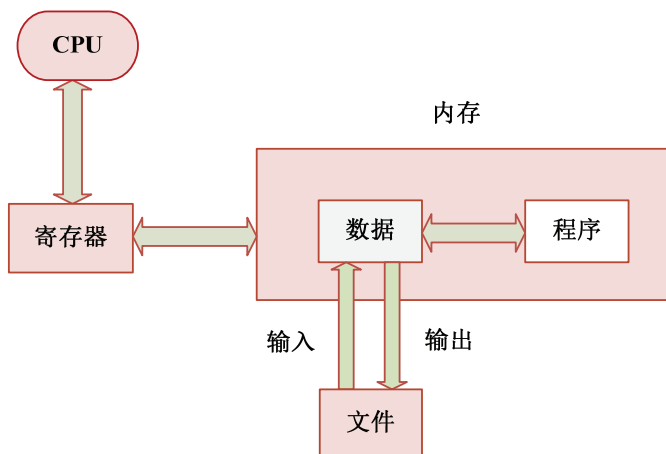


图 18-4 文件处理过程



由于文件存储设备多种多样，各种设备的操作过程差别很大。因此，当代计算机的各种操作系统都对文件及存取文件的各种外围设备进行了抽象。具体原因如图 18-5 所示。

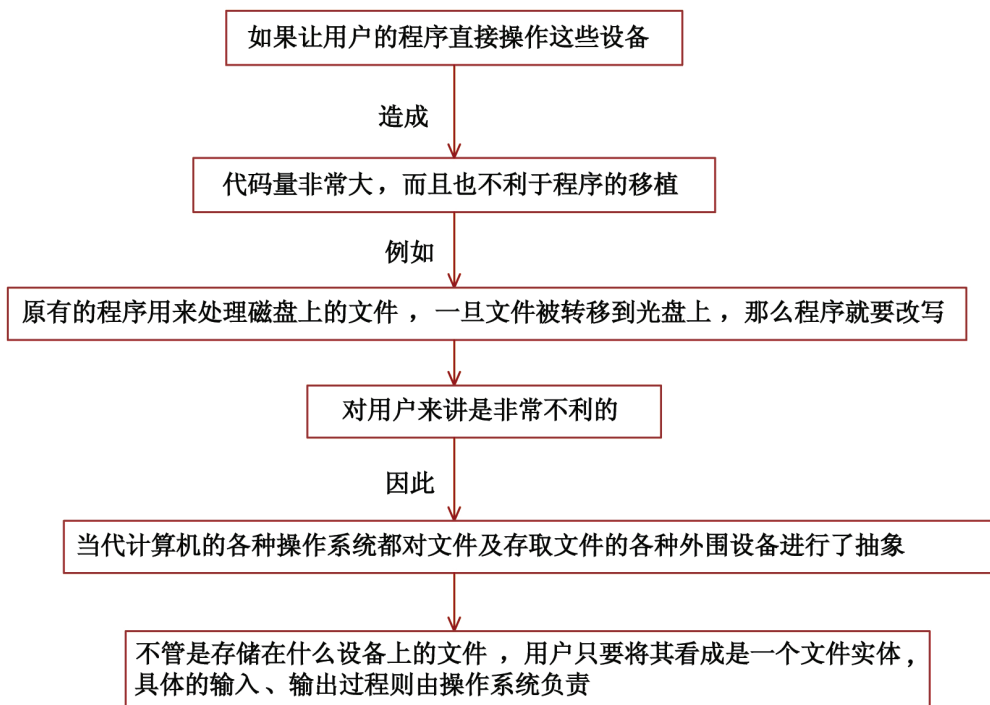


图 18-5 文件进行抽象的原因

面对一个抽象的文件概念，用户程序在处理文件时，一般只需遵从如图 18-6 所示的步骤即可。

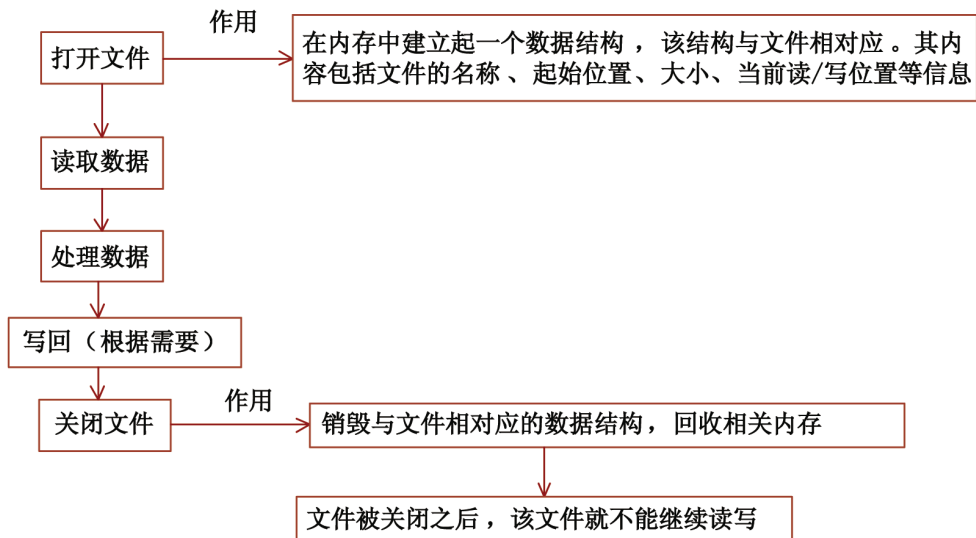


图 18-6 处理文件遵从的步骤



**提示：**关闭文件与打开文件应当一一对应，一个文件被打开，在处理完毕后应当立即关闭。否则会导致内存泄露，更严重的可能会导致文件的完整性和正确性被破坏。

读/写操作是文件处理过程中的关键步骤。读/写之前必须满足一个前提条件，即用户应当了解文件的逻辑结构：文件中存有哪些数据、各种数据的类型是什么、数据之间的顺序结构和关联关系是怎样的等。如果不清楚这些内容，那么就无法完成文件的读/写。即便完成了读/写，那么读取的内容也不会是用户预期的；如果写入，就会很容易破坏文件的完整性和正确性。文件操作的大体过程如图 18-7 所示。

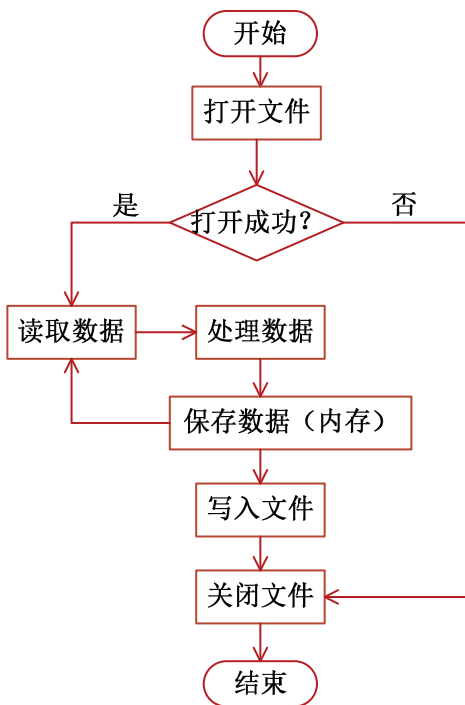


图 18-7 文件操作的过程

由于在对文件进行操作过程中，可能会存在指定的文件不存在或磁盘空间不足等问题，从而导致出错，所以在文件操作的过程中需要注意错误处理。

### 18.1.2 处理文件流的类

C++提供了一系列专门用于处理文件的文件流类。这些类包括专门用于从文件中输入数据的 `ifstream` 类，专门用于向文件输出数据的 `ofstream` 类，以及既可输入也可以输出的 `fstream` 类，如图 18-8 所示。

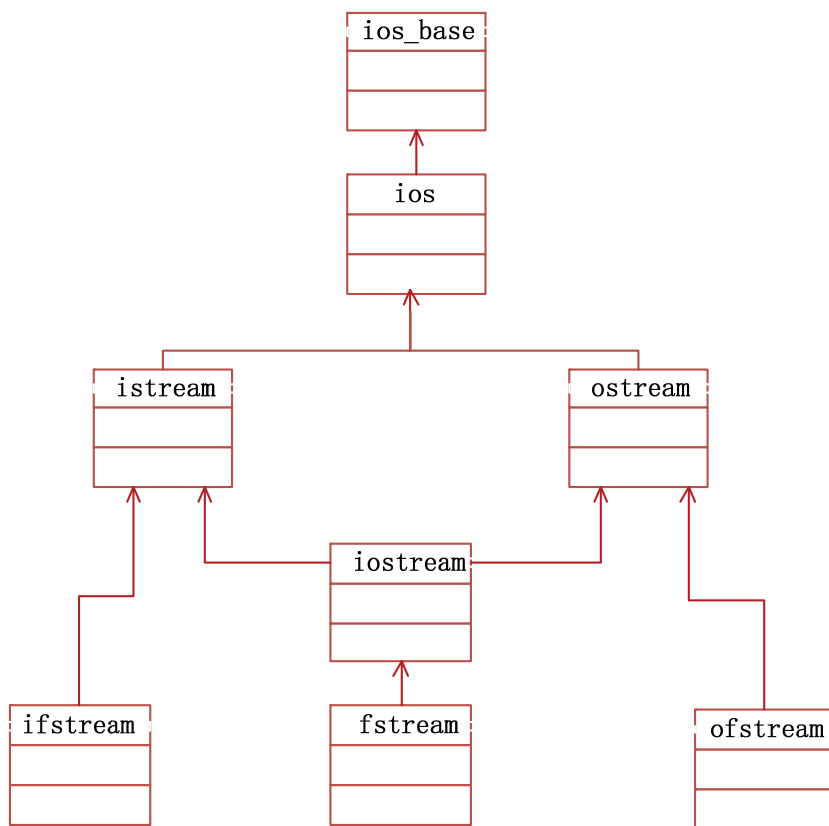


图 18-8 文件流类

文件编码有 ASCII 编码和 Unicode 编码之分，相应的处理文件的类也有这样的区分。上述 ifstream、ofstream 和 fstream 类都是用于处理 ASCII 编码的文件。如果要处理 Unicode 编码的文件，则应当使用对应的 wifstream、wofstream 和 wfstream 类。为简单起见，本书中的文件流类都使用 ASCII 码格式。

【示例 18-1】下面的程序简单展示了使用文件流类编程的过程，其实现代码及结果如图 18-9 所示。



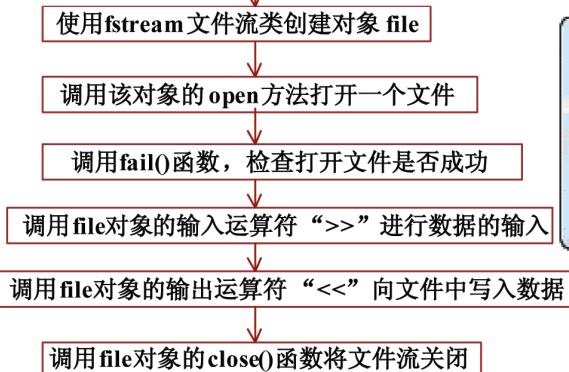
假设在C盘中有一个文件，其文件名为 a.txt，其中的内容为：

```
Hello
World
处理文件
```

实现代码

```
#include <cstdlib>
#include <fstream>
#include <iostream>
using namespace std;
int main(int argc, char* argv[])
{
    fstream file;
    file.open("C:\\a.txt");
    if ( file.fail() )
    {
        cout<<"打开文件失败"<<endl;
        system("PAUSE");
        return 0;
    }
    char str[256];
    while(file >> str )
    {
        cout<< str <<endl;
    }
    file<<"END";
    file.close();
    system("PAUSE");
    return 0;
}
```

执行流程



输出结果

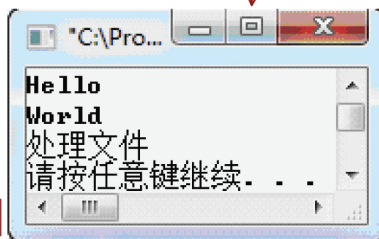


图 18-9 简单文件流类实例



## 18.2 文件的打开与关闭

在 C++ 中, 要进行文件的输入/输出, 必须先创建一个流, 再把这个流与文件相关联 (即打开文件), 才能进行输入/输出操作, 完成后要关闭文件。

C++ 中的 3 个输入/输出流类 `ofstream`、`ifstream` 和 `fstream` 同是 `ios` 的派生类, 可访问在 `ios` 类中定义的所有操作。创建流就是定义流类的对象, 如图 18-10 所示。

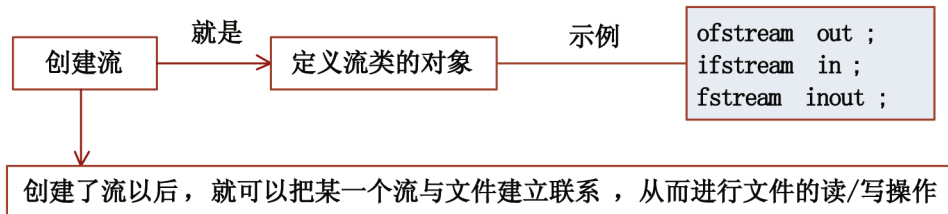


图 18-10 创建流

### 18.2.1 打开文件

打开文件, 就是用函数 `open()` 把某一个流与文件建立联系。`open()` 函数是上述 3 个流类的成员函数, 定义在 `fstream.h` 头文件中, 它的原型如图 18-11 所示。

第二个参数为整型数据类型, 其值决定文件打开的方式, 用数据流基类 `ios_base` 的数据成员指定

```
void open(const unsigned char *, int mode, int access=filebuf::openprot);
```

第一个参数为字符串类型, 用来传递文件名

第三个参数的值决定文件的访问方式及文件的类别, 默认方式是 `filebuf::openprot`

图 18-11 `open()` 函数原型

一般来讲, 每种文件流都有其固定的打开模式, 如图 18-12 所示。

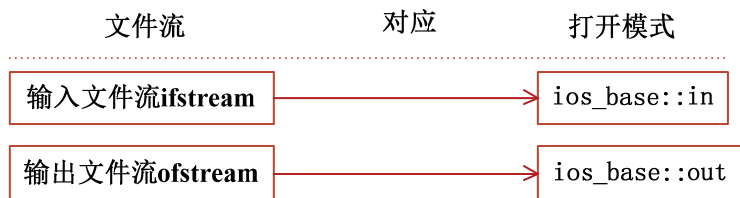


图 18-12 文件流的打开模式

即便是编程时给错了打开模式也不会出问题。例如, 构造输入文件流对象时, 用的打开模式却是 `ios_base::out`。这种情况下并不会影响文件的读入。同时也不要企图向文件中写入数据, 因为输入文件流 `ifstream` 没有输出函数。

在 DOS/Windows 环境中, `access` 的值分别对应 DOS/Windows 的文件属性, 如图 18-13 所示。

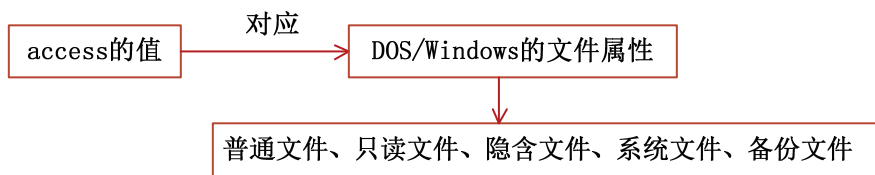


图 18-13 DOS/Windows 的文件属性

【示例 18-2】程序先创建一个文件，然后打开该文件，如果文件不存在，则返回错误信息；否则提示文件已打开的信息。其实现代码及结果如图 18-14 所示。

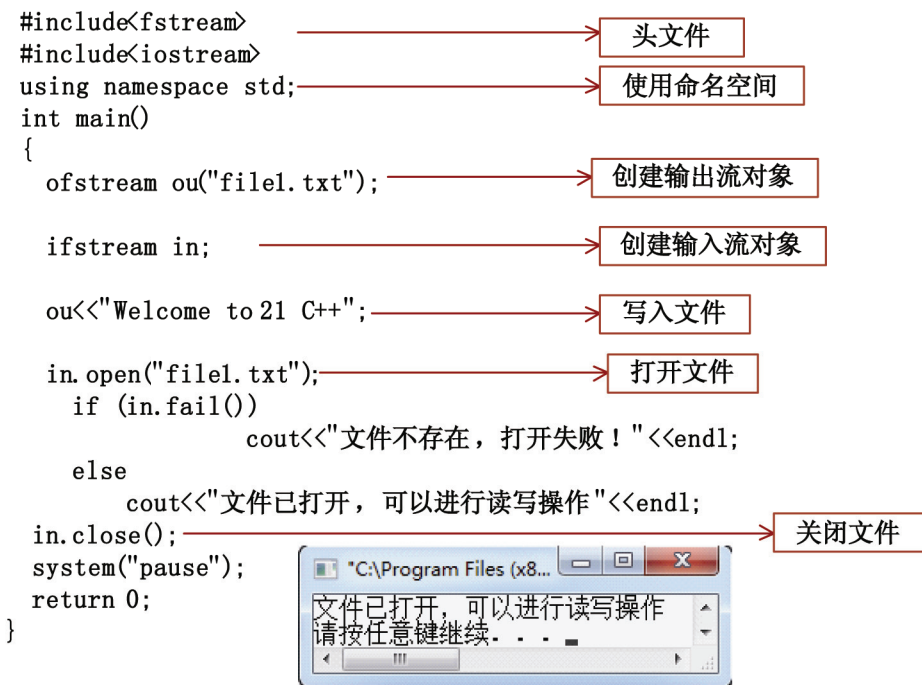
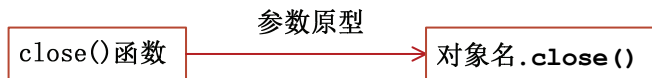


图 18-14 打开文件实例

### 18.2.2 关闭文件

文件使用后，必须将其关闭，否则可能导致数据的丢失。关闭文件就是将文件与流的联系断开，用函数 `close()` 完成，它也是流类中的成员函数，没有参数，没有返回值，其参数原型如图 18-15 所示。

图 18-15 函数 `close()` 的参数原型

【示例 18-3】下面的程序使用 `open()` 函数打开一个文件，并判断其是否打开，但不管其是否打开成功，在程序结束时均需要关闭文件。其实现代码及结果如图 18-16 所示。



```
#include<fstream>
#include <iostream>
using namespace std;
int main()
{
    ifstream in;
    in.open("file2.txt");
    if (in.fail())
        cout<<"文件不存在，打开失败！"<<endl;
    else
        cout<<"文件已打开，可以进行读写操作"<<endl;
    in.close();
    cout<<"文件已关闭"<<endl;
    system("pause");
    return 0;
}
```

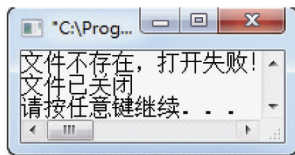


图 18-16 关闭文件实例

输入打开文件的路径时，反斜杠要双写，因为编译器认为反斜杠是转义字符标志。如“C:\\myfile”表示 C 盘下的 myfile 文件。

在程序语句中，可将定义流与打开文件用一条语句完成，如图 18-17 所示。

示例	
定义流与打开文件用一条语句	<pre>ofstream out ("test", ios::out, 0); fstream io ("test", ios::in ios::out, 0);</pre>

图 18-17 定义流与打开文件用一条语句完成



**注意：**一般情况下，ifstream 和 ofstream 流类的析构函数可以自动关闭已打开的文件，但若需要使用同一个流对象打开文件，则需要首先用 close() 函数关闭当前文件。



## 18.3 文件的顺序读/写

文件的读/写主要包括顺序读/写和随机读/写两种方式，本节将主要讲解文件顺序读/写的实现方法。

### 18.3.1 读/写文本文件

对文本文件进行读/写时，先要以某种方式打开文件，然后使用运算符“<<”和“>>”进行操作，同时必须将运算符“<<”和“>>”前的 cin 和 cout 用与文件相关联的流代替。例如，cin 可以用 fin 流代替，cout 可以用 fout 代替等。

【示例 18-4】下面的程序先向文本文件 test.txt 中写入 3 行数据，再将该文件打开，并将写入的数据依次输出到屏幕上。其实现代码及结果如图 18-18 所示。



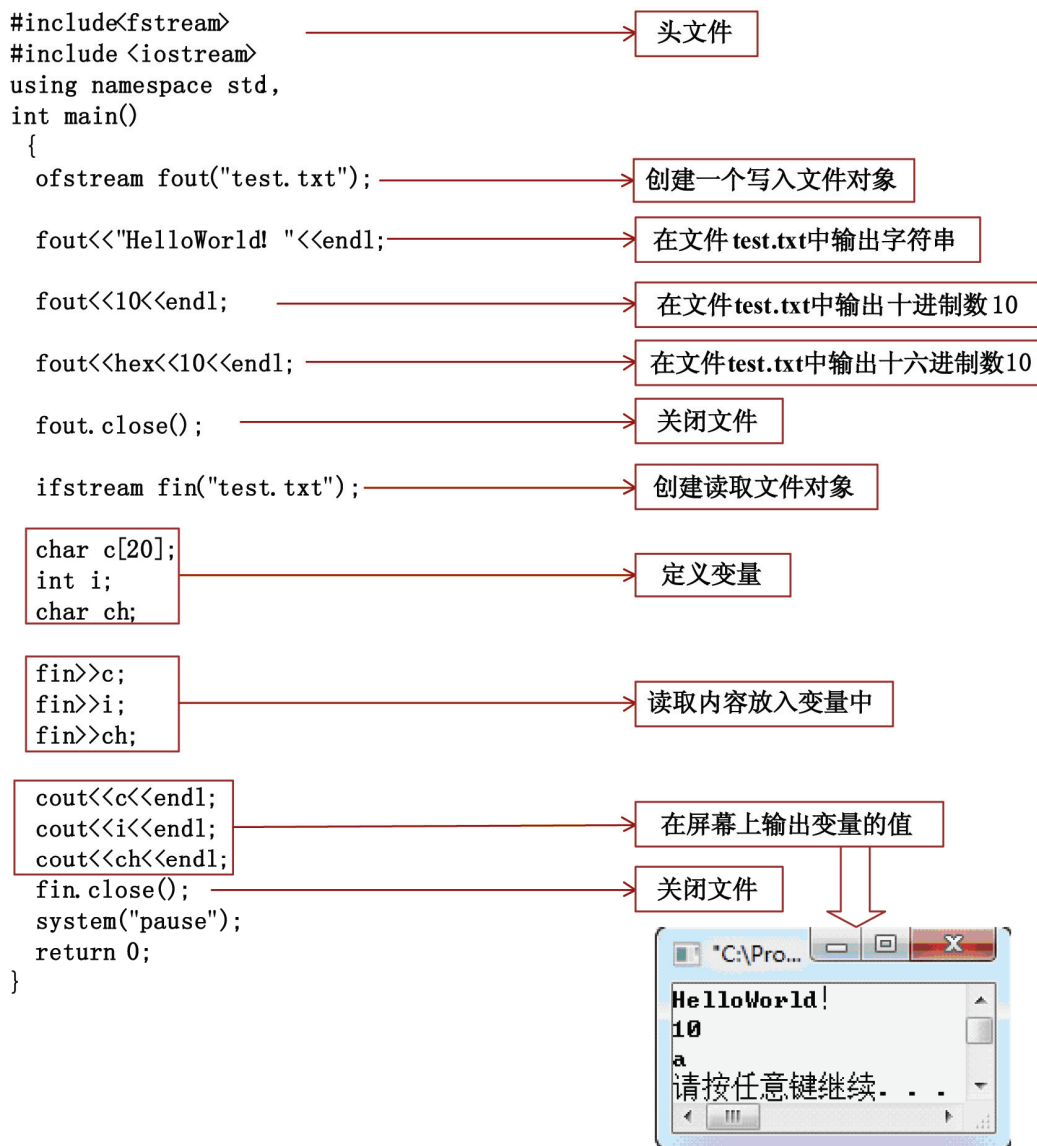


图 18-18 读写文本文件实例

### 18.3.2 读/写二进制文件

二进制文件是一种不能用普通的字处理软件进行编辑、占空间较小的文件。二进制文件与文本文件的区别，如图 18-19 所示。



不同点	文本文件	二进制文件
① 本质	文本文件是字符流	二进制文件是字节流
② 输入/输出转换	输入时，将回车和换行两个字符转换为字符“\n”，输出时将字符“\n”转换为回车和换行两个字符，	不做转换
③ 遇到文件结束符	用get()函数返回一个文件结束标志EOF，该标志的值为-1	用成员函数eof()判断文件是否结束

图 18-19 二进制文件与文本文件的区别

任何文件都能以文本方式或二进制方式打开。对以二进制方式打开的文件，有两种方式进行读/写操作，如图 18-20 所示。

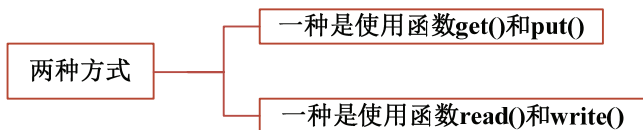


图 18-20 二进制方式打开文件的两种方式

### 1. 使用函数 get()和 put()读/写二进制文件

get()函数是输入流类 istream 中定义的成员函数，其作用及实现的功能如图 18-21 所示。

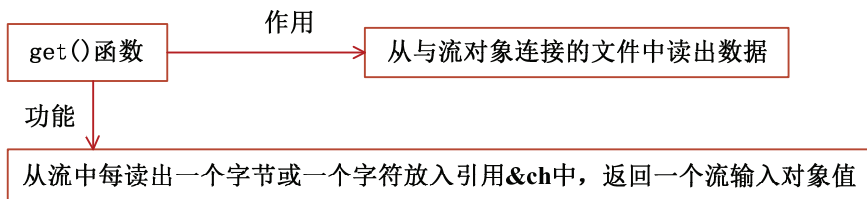


图 18-21 get()函数的作用及实现功能

在 C++中，get()函数的原型如图 18-22 所示。



图 18-22 get()函数的原型

put()函数是输出流类 ostream 中定义的成员函数，其作用及实现功能如图 18-23 所示。

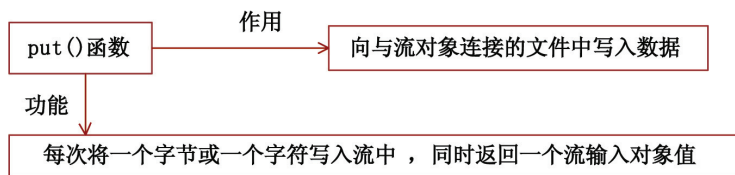


图 18-23 put()函数的作用及实现功能



在 C++ 中，put() 函数的原型如图 18-24 所示。

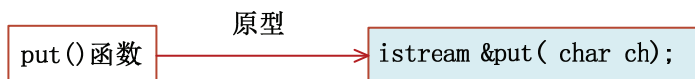


图 18-24 put() 函数的原型

get() 函数和 put() 函数都只能对单个字符或单个字节进行操作，如果实现多字符的操作，可通过循环语句来实现。

【示例 18-5】下面的程序用函数 get() 和 put() 读/写二进制文件，其实现代码及结果如图 18-25 所示。

```
#include<fstream>
#include <iostream>
using namespace std;
main(int argc, char *argv[]) ➔ 带参数的 main() 函数
{
    char ch;
    if (argc!=3) ➔ 参数个数错误
    {
        cout<<"命令行输入错误！"<<endl;
        return 1;
    }

    ifstream fin(argv[1]); ➔ 定义输入流对象，打开第二个参数中的文件
    if(!fin)
    {
        cout<<"不能打开源文件。"<<endl;
        return 1;
    }

    ofstream fout(argv[2]); ➔ 打开第三个参数中的文件
    if(!fout)
    {
        cout<<"不能打开目标文件。"<<endl;
        return 1;
    }

    while(fin) ➔ 循环写入
    {
        fin.get(ch); ➔ 从源文件中读出字符
        fout.put(ch); ➔ 写入到目标文件中
    }

    fin.close(); ➔ 关闭流
    fout.close();
    system("pause");
    return 0;
} ➔ 带参数的main()函数，需要参数才能运行，因此直接运行上述程序，将导致执行错误
```



图 18-25 使用函数 get() 和 put() 读写二进制文件实例



## 2. 使用函数 read()和 write()读/写二进制文件

C++支持使用函数 read()和 write()读/写二进制文件。read()函数是输入流类 istream 中定义的成员函数，其常用的原型及作用如图 18-26 所示。

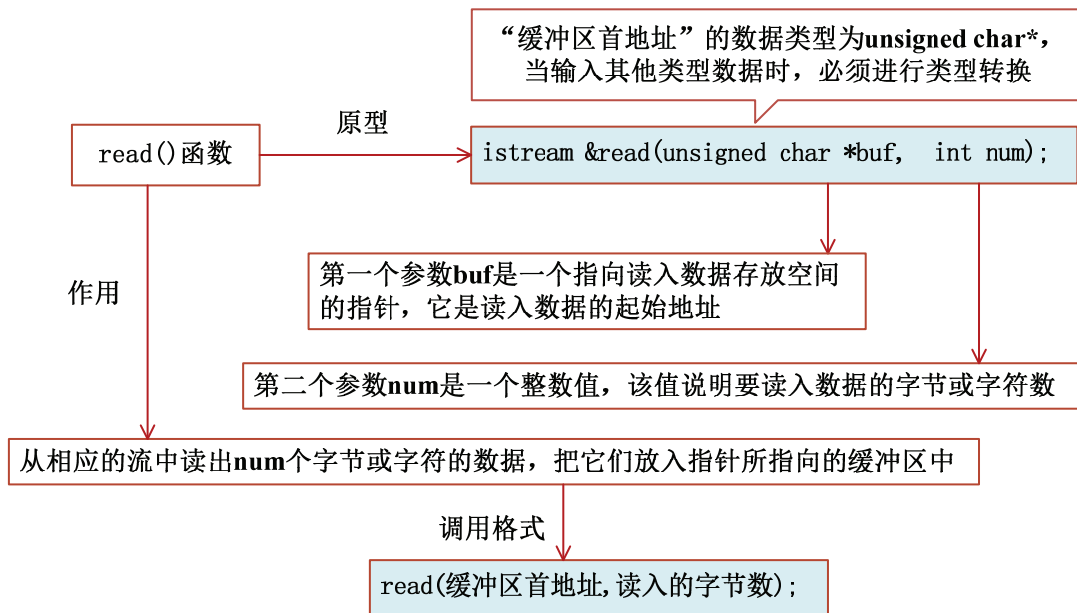


图 18-26 read()函数原型及作用

write()函数是输出流类 ostream 中定义的成员函数，其常用的原型及作用如图 18-27 所示。

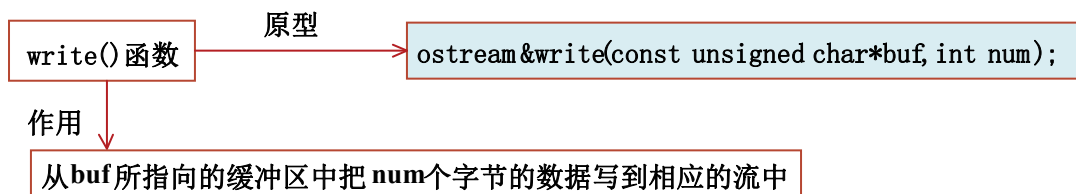


图 18-27 write()函数原型及作用

write()函数的参数的含义、调用及注意事项与函数 read()相同。

【示例 18-6】下面的程序用函数 read()和 write()读/写二进制文件，实现向文件 test.txt 写入一个双精度数据类型的数值和一个字符串，并将该文件中的内容读出显示到屏幕中。其实现代码及结果如图 18-28 所示。

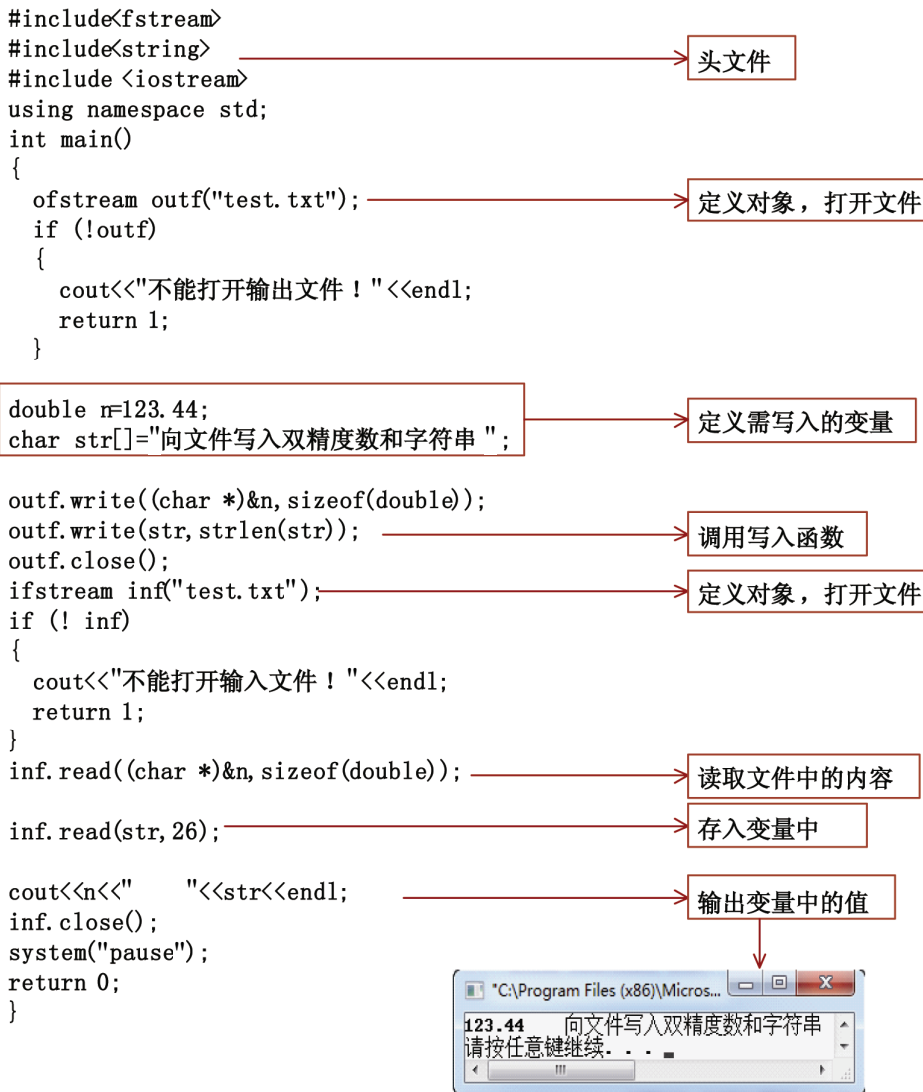


图 18-28 使用函数 read()和 write()读/写二进制文件实例

其实，上述 get()、put()、read()和 write()函数也可以用于文本文件，其处理过程与二进制文件的处理过程基本相同，这里不再介绍。

## 18.4 文件的随机读/写

前面介绍的有关文件的读/写操作，都是按一定的顺序进行读/写的，称为顺序文件，其特点是只能按数据在文件中的排列顺序一个一个地访问数据，使用很不方便。为此，C++又提供了文件的随机读/写。

随机读/写是通过使用输入或输出流中与随机移动文件指针相关的成员函数，通过随意移动文件指针而达到随机访问的。移动文件指针的成员函数主要有 seekg()和 seekp()，其常用原型如图 18-29 所示。

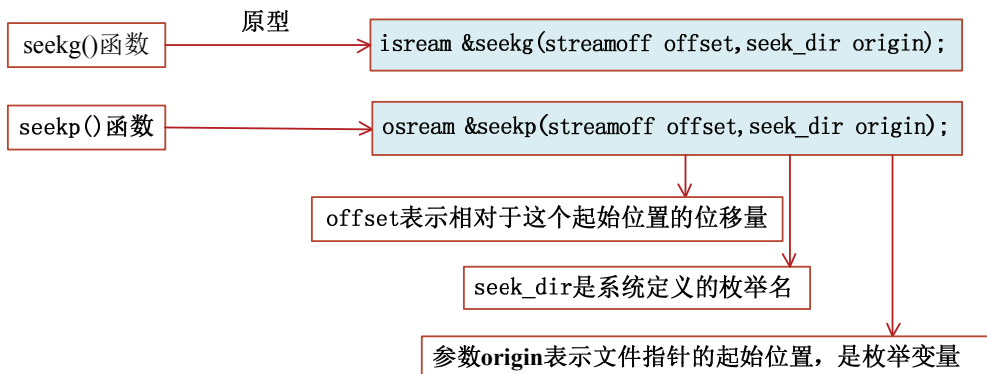


图 18-29 seekg()和 seekp()函数原型

origin 的取值有 3 种情况，如图 18-30 所示。

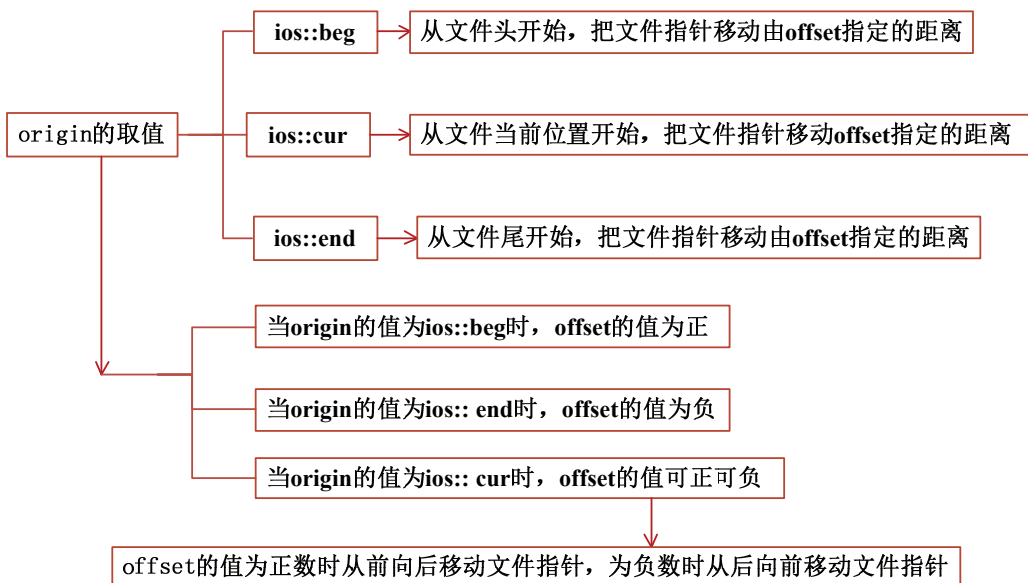


图 18-30 origin 的取值

位移量 offset 的类型是 streamoff，该类型为一个 long 型数据，它在头文件 iostream.h 中定义，如图 18-31 所示。



图 18-31 streamoff 在头文件 iostream.h 中定义

此外，移动文件指针的成员函数 seekg()和 seekp()的使用范围和功能如图 18-32 所示。

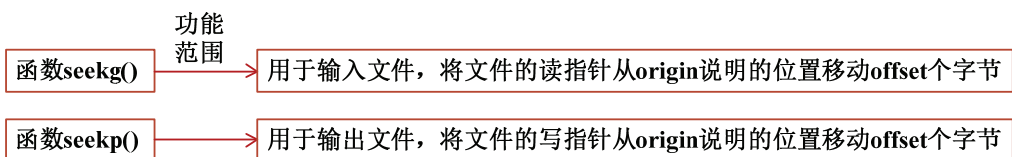


图 18-32 成员函数 seekg()和 seekp()的使用范围和功能



进行文件的随机读/写时，可用下列函数确定文件当前指针的位置，如图 18-33 所示。

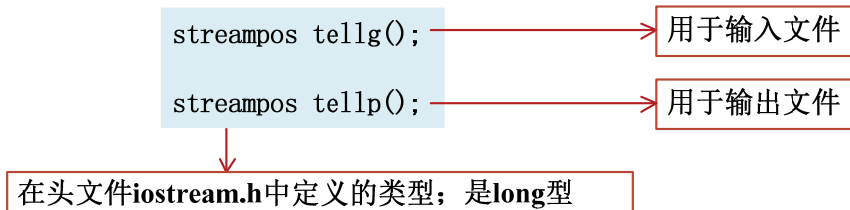


图 18-33 确定文件当前指针位置的函数

【示例 18-7】下面的程序可实现文件的随机读/写，其实现代码及结果如图 19-34 所示。

```
#include<fstream>
#include<stdlib.h>
#include<iostream>
using namespace std;
main(int argc,char *argv[])
{
    char ch;
    if (argc !=3)
    {
        cout<<"命令行输入错误！"<<endl;
        return 1;
    }
    ifstream fin(argv[1]);
    if (!fin)
    {
        cout<<"不能打开输入文件！"<<endl;
        return 1;
    }
    fin.seekg(atoi(argv[2]), ios::beg);
    while(!fin.eof())
    {
        fin.get(ch);
        cout<<ch;
    }
    fin.close();
    system("pause");
    return 0;
}
```

带参数的 `main()` 函数

参数个数不为 3

定义输入流并打开源文件

将依据参数 3 的值，从文件头偏移指定的距离

逐个字符读取

输出到屏幕上

第二个参数是该程序要访问的数据文件，如果不在当前目录下，需要给出完整路径

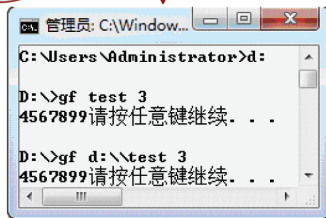


图 18-34 文件的随机读/写实例



## 18.5 小结

本章主要介绍了 C++ 中文件的概念及其相关的操作。对于任意一个文件在进行操作前，都必须打开文件，然后进行文件的读/写操作；最后需要关闭文件。接着通过实例详细介绍了文本文件、二进制文件和随机文件的读/写操作。



## 18.6 习题

【题目 18-1】根据用户输入的文件名和路径，读取指定的文本文件内容并显示出来。例如，文件 test.dat 中的内容如图 18-35 所示，程序运行结果如图 18-36 所示。

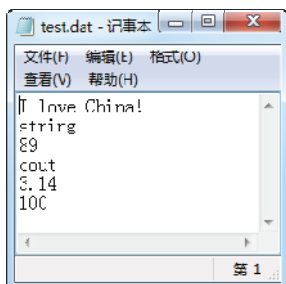


图 18-35 文件内容

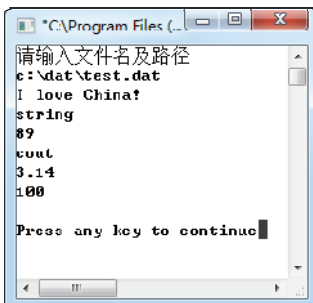


图 18-36 运行结果

【题目分析】本题要求读者熟悉 C++ 文件操作流的相关知识，重点是掌握文件操作流的使用。

### 【关键代码】

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
    char filename[FILENAME_MAX]={0};
    char str[FILENAME_MAX]={0};
    cout<<"请输入文件名及路径"<<endl;
    cin>>filename;
    ifstream in(filename);
    if(!in.is_open())
    {
        cout<<"读取文件失败";
    }
    while(!in.eof())
    {
        in.read(str,FILENAME_MAX-1);
        cout<<str<<endl;
    }
    in.close();
    return 0;
}
```

【题目 18-2】创建文件 test.dat，在其中写入 5、3.1415926、string 字符。然后从 test.dat 文件中将写入的信息读出，赋给相应的变量，并借助标准输出/输入流对象 cout 输出到显示器显示。程序的运行效果如图 18-37 所示。



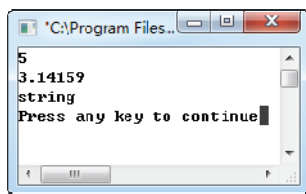


图 18-37 运行结果

【题目分析】本题要求读者熟悉 C++ 文件操作流的相关知识。

【关键代码】

```
ofstream out("test.dat");

if(!out.is_open())
{
    cout<<"文件打开失败";
    return 0;
}

out<<5<<" "<<3.1415926<<" "<<"string\n";
out.close();

int ix;
double dy;
char sz[10];
ifstream in("test.dat");

if(!in.is_open())
{
    cout<<"文件读取失败";
}

in>>ix>>dy>>sz;
cout<<ix<<endl<<dy<<endl<<sz<<endl;
in.close();
```



· 轻松学开发 ·



本书技术支持 [www.rzchina.net](http://www.rzchina.net)

上架建议：程序开发 > C++



@博文视点Broadview



策划编辑：胡辛征  
责任编辑：高洪霞  
封面设计：侯士卿

ISBN 978-7-121-19809-0



9 787121 198090 >

定价：55.00元（含DVD光盘1张）